Appendix B. Program Listing

```
/*
 ***********************************************************************
 *                                                                     *
 * Program to control LORAN-C radio                                    *
 *                                                                     *
 * This program controls a special-purpose radio designed to receive   *
 * transmissions from the US Coast Guard LORAN-C navigation system.    *
 * These stations operate on an assigned radio frequency of 100 kHz    *
 * and can be received over the continental US, adjacent coastal areas *
 * and significant areas elsewhere in the world.                       *
 *                                                                     *
 * The radio, which is contained in a separate, shielded box, receives *
 * the signals, which consist of an eight-pulse biphase-modulated      *
 * pulse group transmitted at a 1-kHz rate. Each of these pulse groups *
 * is repeated at an interval characteristic of the particular LORAN-C *
 * chain, which consists of a master station and up to three slave     *
 * stations. The radio includes a synchronous detector driven by a     *
 * quadrature-phase clock, two integrators with adjustable gain and    *
 * an signal-level detector used to derive the agc voltage.            *
 *                                                                     *
 * The radio is controlled by this program using a special-purpose     *
 * interface, which converts the receiver signals using an             *
 * analog/digital converter and multiplexor, and generates the digital *
 * timing and analog control signals using an AMD 9513A System Timing  *
 * Controller (STC) chip, two digital/analog converters and            *
 * miscellaneous logic components. The radio provides three analog     *
 * signals, one for the in-phase integrator, another for the           *
 * quadrature-phase integrator and a third for the agc. This program   *
 * computes the master oscillator frequency adjustment and receiver    *
 * agc voltage.                                                        *
 *                                                                     *
 * The reciever includes a precision, oven-controlled crystal          *
 * oscillator used to derive all timing signals used by the receiver   *
 * and this program. The 5-MHz output of this oscillator is adjusted   *
 * over a small range by this program to coincide with the LORAN-C     *
 * signal as broadcast to within a few parts in 10e10 and is suitable  *
 * for use as a laboratory frequency standard. The oscillator itself   *
 * should have good intrinsic stability and setability to within less  *
 * than 2.5 Hz at 5 MHz (0.5 ppm), since it must maintain the master   *
 * clock to within 100 us over the pulse-code scan interval of several *
 * minutes.                                                            *
 *                                                                     *
 * The PC running this program generates the control signals necessary *
 * to run the radio and produces a 1-pps signal synchronized to        *
 * UTC(LORAN) to within a fraction of a microsecond. When manually     *
 * adjusted using time-of-coincidence (TOC) data published by US Naval *
 * Observatory, this signal is suitable for use as a precision source  *
 * of standard time. The system can generate all sorts of external     *
 * signals as well, as programmed in the 9513A.                        *
 *                                                                     *
 *      David L. Mills (mills@udel.edu) 27 March 1992                  *
 *                                                                     *
 ***********************************************************************
 *
```

```
 * Current LORAN-C chains by gri (master listed first)
 *
 * 9990 North Pacific        St. Paul Island, Attu, Port Clarence,
 *                                Narrow Cape
 * 9980 Icelandic Sea
 * 9970 North West Pacific   Iwo Jima, Marcus Island, Hokkaido, Gesashi,
 *                                Yap Island
 * 9960 North East US        Seneca, Caribou, Nantucket, Carolina Beach,
 *                                Dana
 * 9940 West Central US      Fallon, George, Middletown, Searchlight
 * 9610 South Central US
 * 8970 Great Lakes US       Dana, Malone, Seneca, Baudette
 * 8290 North Central US
 * 7990 Mediterranian Sea    Sellia Marina, Lampedusa, Kargabarun,
 *                                Estartit
 * 7980 South East US        Malone, Grangeville, Raymondville, Jupiter,
 *                                Carolina Beach
 * 7970 Norwegian Sea        Ejde, Sylt, B0, Sandur, Jan Mayen
 * 7960 Gulf of Alaska       Tok, Narrow Cape, Shoal Cove
 * 7930 Labrador Sea         Angissoq, Sandur, Ejde, Cape Race
 * 5990 West Central Canada  Williams Lake, Shoal Cove, Port Hardy
 * 5930 East Central Canada  Caribou, Nantucket, Cape Race
 * 4990 Central Pacific      Johnson Island, Upolu Point, Kure Island
 */

#include <stdio.h>
#include <ctype.h>
#include <bios.h>
#include <math.h>
#include <conio.h>
#include <string.h>

/*
 * Sizes of things. The pulse-group filter is a shift register with one
 * stage for each 100-us bin in a 1000-us sample window plus two stages
 * for a noise gate. The envelope filters consist of one stage for each
 * 10-us cycle in the 300-us envelope pulse gate, for a total of 30
 * samples.
 */
#define NRMS 10                        /* size of pulse-group filter */
#define NENV 30                        /* size of envelope filter */

/*
 * Program characteristics. The field and display guard times are the
 * maximum latency for the program to process samples at the end of a
 * gri and for the output routines to print a line, respectively. If the
 * time to the next gri is less than either of these, the next frame is
 * skipped. The watchdog timeout is the maximum number of frames before
 * the program abandons cycle search and reverts to pulse-group seach.
 * The agc averaging factor sets the time constant of integration for
 * the receiver signal- level indicator. The remaining parameters
 * establish the minimum and maximum median filter size and envelope and
 * phase weights.
 */
#define FGUARD 1000               /* field guard time (100 us) */
```

```
#define DGUARD 8000               /* display guard time (100 us) */
#define WATCHDOG 2000             /* watchdog timeout (frame) */
#define AGCFAC 16                 /* agc averaging factor */
#define MMIN 3                    /* min median filter size */
#define MMAX 10                       /* max median filter size */
#define EMIN 5                    /* min envelope weight */
#define EMAX 50                       /* max envelope weight */
#define EFAC 1.1                  /* envelope adjustment factor */
#define PMIN 5                    /* min phase weight */
#define PMAX 200                  /* max phase weight */
#define PFAC 1.2                  /* phase adjustment factor */

/*
 * Receiver characteristics. The receiver delay is characteristic of the
 * receiver bandpass. The pulse-group offset is characteristic of the
 * pulse-group filter and noise gate.
 */
#define RCVDELAY 50               /* receiver delay (200 ns) */
#define OFFSET 21                 /* pulse-group offset (10 us) */

/*
 * Receiver gain and noise gates. The vco parameter is adjusted
 * for zero nominal frequency offset. The agc parameter is adjusted for
 * nominal receiver output on the peak loran pulse of 800 mv p-p. The
 * receiver gain parameter is adjusted so that the agc threshold (knee)
 * occurs at a peak signal amplitude of 100, as determined by the status
 * display. The derived envelope factor is adjusted so the zero crossing
 * of the derived envelope signal occurs at the third cycle. The noise
 * gate parameters establish the error/false-alarm rates.
 */
#define VCO 194                       /* initial vco dac */
#define AGC 162                       /* initial agc dac */
#define RGAIN 2.5                 /* receiver gain */
#define DERVEL 2.2908             /* derived envelope factor */
#define PGATE 3                       /* pulse-group noise gate */
#define SGATE 2                       /* strobe noise gate */

/*
 * The receiver agc is controlled to produce a q-channel amplitude of
 * 100, which represents a demodulator transfer function at the third
 * carrier cycle of 50 V/rad. The vco has a sensitivity of 1 Hz/V
 * reduced to 0.1 Hz/V by a pad between the dac and the vco, which
 * represents a transfer function of 0.628 rad/V-s. The dac produces 6 V
 * p-p for an input of 256V p-p, for a transfer function of .0234. The
 * ratio of the 100-kHz demodulator clock to the vco frequency (5 MHz)
 * is 1/50. The overall pll gain is the product of these factors .0147,
 * rounded up to .015 for neatness.
 */

#define VGAIN .015                /* overall loop gain */

/*
 * Timing generator definitions
 */
#define PORT 0x0300               /* controller port address */
```

```
#define TGC PORT+0                /* stc control port (r/w) */
#define TGD PORT+1                /* stc data port (r/w) */
/*
 * Analog/digital converter definitions
 */
#define ADC PORT+2                /* adc buffer (r)/address (w) */
#define ADCGO PORT+3              /* adc status (r)/adc start (w) */
     #define START 0x01           /* converter start bit (w) */
     #define BUSY 0x01            /* converter busy bit (r) */
     #define DONE 0x80            /* conversion done bit (r) */
/*
 * Digital/analog converter definitions
 * Note: output voltage is inverted from buffer
 */
#define DACA PORT+4               /* vco (dac a) buffer (w) */
#define DACB PORT+5               /* agc (dac b) buffer (w) */
/*
 * Code generator definitions
 * Note: bits are shifted out from the lsb first
 */
#define CODE PORT+6               /* pulse-code buffer (w) */
     #define MPCA 0xCA            /* LORAN-C master pulse code group a */
     #define MPCB 0x9F            /* LORAN-C master pulse code group b */
     #define SPCA 0xF9            /* LORAN-C slave pulse code group a */
     #define SPCB 0xAC            /* LORAN-C slave pulse code group b */
/*
 * Mode register definitions
 */
#define PAR PORT+7                /* parameter buffer (w) */
     #define INTEG 0x03           /* integrator mask */
     /*
      * time constant values
      *
      * 0 1.000 ms
      * 1 0.264 ms
      * 2 0.036 ms
      * 3 short caps
      */
     #define GATE 0x0C            /* gate source mask */
     /*
      * gate source values
      *
      * 4 always open
      * 8 group repeition interval (GRI)
      * c puldse code interval (PCI)
      * f strobe (STB)
      */
     #define IEN 0x20             /* enable interrupt bit */
     #define EN5 0x40             /* enable counter 5 bit */
     #define ENG 0x80             /* enable gri bit */

/*
 * Timing generator (STC) commands
 */
/* argument sssss = counter numbers 5-1 */
```

```
#define LOADDP 0x00               /* load data pointer */
     /* argument ee = element (all groups except ggg = 000 or 111) */
     #define MODEREG 0x00         /* mode register */
     #define LOADREG 0x08         /* load register */
     #define HOLDREG 0x10         /* hold register */
     #define HOLDINC 0x18         /* hold register (hold cycle increm) */
     /* argument ee = element (group ggg = 111) */
     #define ALARM1 0x07          /* alarm register 1 */
     #define ALARM2 0x0F          /* alarm register 2 */
     #define MASTER 0x17          /* master mode register */
     #define STATUS 0x1F          /* status register */
#define ARM 0x20                  /* arm counters */
#define LOAD 0x40                 /* load counters */
#define LOADARM 0x60              /* load and arm counters */
#define DISSAVE 0x80              /* disarm and save counters */
#define SAVE 0xA0                 /* save counters */
#define DISARM 0xC0               /* disarm counters */
/* argument nnn = counter number */
#define SETTOG 0xE8               /* set toggle output HIGH for counter */
#define CLRTOG 0xE0               /* set toggle output LOW for counter */
#define STEP 0xF0                 /* step counter */
/* argument eeggg, where ee = element, ggg - counter group */
/* no arguments */
#define ENABDPS 0xE0              /* enable data pointer sequencing */
#define ENABFOUT 0xE6             /* enable fout */
#define ENAB8 0xE7                /* enable 8-bit data bus */
#define DSABDPS 0xE8              /* disable data pointer sequencing */
#define ENAB16 0xEF               /* enable 16-bit data bus */
#define DSABFOUT 0xEE             /* disable fout */
#define ENABPFW 0xF8              /* enable prefetch for write */
#define DSABPFW 0xF9              /* disable prefetch for write */
#define RESET 0xFF                /* master reset */

/*
 * Function declarations
 */
void status(double, double, char*);

/*
 * STC setup. Note gri = 99600, pci = 300 and stb = 10 (us).
 *
 * Counter 1 generates a 200-kHz signal from the 5-MHz master VCO. This
 * signal is a slightly assymetrical square wave (duty factor 12/13). An
 * external flipflop divides this signal by two to get the 100-kHz gri
 * clock which drives counter 2. All other counters are driven from the
 * 5-MHz source. The 200-kHz and 100-kHz signals are used by the
 * synchronous demodulator in the receiver.
 *
 * Counter 2 generates the gri (pulse-code) gate, which repeats at the
 * interval assigned to the LORAN-C chain; e.g., 9960 for the Northeast
 * U.S. chain. The signal consists of a high 8-ms interval preceeded by
 * a programmable low interval normally equal to the gri interval less 8
 * ms. Counter 3 generates the pulse-code (pci) gate, which enables the
 * receiver only when a pulse group is expected. The signal consists of
 * eight 300-us high intervals beginning at the high interval of counter
```

```
 * 2. Counter 4 generates the stb (cycle) gate used during envelope
 * scan. The signal consists of a high 10-us interval preceeded by a
 * programmable interval in the range up to about 300 us.
 *
 * Counter 5 operates as a gated divider to drive the frequency scalar
 * and output divider, which produces the output signals for external
 * equipment. The gating signal is generated by counter 4, which can be
 * enabled for this purpose under probram control. When so enabled,
 * counter 5 is stopped for the interval programmed in counter 4,
 * enabling precise alignment of the frequency scalar and output divider
 * to UTC. The output signal can be at 1 pps and any decimal multiple up
 * to 100 kHz plus 5 MHz or, if UTC alignment is not necessary, any
 * binary or decimal submultiple of 5 MHz. Note that all counters
 * operate in binary mode, except the frequency scalar and output
 * divider, which normally operate in bcd mode.
 */
int init[] = {
     0x0162,   12,   13,          /* counter 1 (p0) */
     0x0262, 9160,  800,          /* counter 2 (gri) */
     0x8162, 1500, 3500,          /* counter 3 (pcx) */
     0xc162,    0,   50,          /* counter 4 (stb) */
     0x0162,   25,   25           /* counter 5 (out) */
     };
/*
 * Standard envelope cycle amplitudes. These are matched with the
 * received envelope amplitudes to compute the rms error, assuming the
 * reference cycle (3) is at the reference phase in the envelope window.
 * Following are the cycle amplitude values (peak-normalized to 100) in
 * the standard LORAN transmission specification.
 *
 *        cycle     1     2     3*    4     5     6     7 */
double envcyc[7] = {4.7, 25.3, 50.8, 72.9, 88.6, 97.3,100.0};

/*
 * Program variables (units)
 */
int ptrenv = 0;                        /* index of display cycle */
char mode = '1';                  /* operating mode */
char pulse = 'm';                 /* master (m) or slave (s) codes */
int par = 0x0a;                        /* mode register */
char kbd = ' ';                        /* latest keystroke */

/*
 * System timing and rrelated data (units). These values provide the
 * precise offset of the reference phase relative to the epoch when
 * the STC chip was last reset by this program. All timing calculations
 * are performed relative to this epoch. The freq and phase variables
 * are computed in various places in the program, but take effect only
 * at the end of the processing cycle, so that all timing calculations
 * can be performed with respect to the epoch the system is actually at.
 */
int frame = 0;                    /* offset to reference frame (2*gri) */
int offset = 0;                        /* offset to reference gri (10 us) */
int strobe = 0;                        /* offset to reference cycle (10 us) */
int gri = 9960;                        /* group repetition interval (10 us) */
```

```
int pcx = 1500;                         /* pulse-code interval (200 ns) */
int freq = 0;                      /* frequency offset (10 us/frame) */
int phase = 0;                     /* phase offset (10 us) */
int step = 1;                      /* phase step (10 us) */
int stb[MMAX];                     /* strobe median filter */
int sgate = SGATE;                 /* strobe noise (max-min) */

/*
 * Various controls normally preset, but can be adjusted by keyoard
 * commands. After all, this is a prototype device.
 */
double vco = VCO;                  /* vco dac signal */
double vcodac = VCO;               /* vco dac bias (dac a) */
double agc = AGC;                  /* agc dac signal */
double agcdac = AGC;               /* agc dac bias (dac b) */

/*
 * Raw and processed data input from receiver. The raw data are received
 * directly from the adc and summed for both the a and b gri intervals.
 * The offset data are computed during the receiver calibration modes
 * and used to remove bias from the raw data to produce the net signal
 * valid at the end of the frame. As the envelope pointer cycles through
 * all 30 100-us cycles of the pulse gate, the i and q signal envelopes
 * are averaged separately for use in the cycle-identification and
 * phase-tracking processes. The median filters are used to suppress
 * impulse-noise and pulse-dropout. The rms error signal produced from
 * the i and q signals is used during the pulse-code identification
 * process.
 */
double isig = 0;                   /* i-signal (a+b) */
double qsig = 0;                   /* q-signal (a+b) */
double pmed[NENV][MMAX];           /* phase median filter */
double emed[NENV][MMAX];           /* envelope median filter */
double mgate = 0;                  /* envelope median filter span */

double pcyc[NENV];                 /* cycle phase */
double pfac = PMIN;                /* cycle phase weight */
double ecyc[NENV];                 /* cycle amplitude */
double efac = EMIN;                /* cycle amplitude weight */
double erms[NENV];                 /* cycle rms error */
double edrv[NENV];                 /* cycle derived envelope */

double iofs = 307;                 /* i-integrator offset */
double qofs = 308;                 /* q-integrator offset */
double agcraw = 0;                 /* receiver agc output (adc chan 2) */
double agcavg = 0;                 /* receiver agc smoothed signal */
double agcofs = 317;               /* receiver agc offset (zero signal) */
double agcmax = 355;               /* receiver agc max signal (overload) */
double rms[NRMS];                  /* pulse-group signal shift register */
double gain = RGAIN;               /* program gain */
double dervel = DERVEL;            /* derived envelope factor */
char report[257] = "\0";           /* report string for display */

/*
 * Event counters. These tally the synchronization events of interest
```

```
 * for debugging and monitoring.
 */
int pgcnt = 0;                    /* pulse-group search events */
int encnt = 0;                    /* envelope search events */
int cscnt = 0;                    /* cycle-slip events */
int sscnt = 0;                    /* strobe-slip events */
int pncnt = 0;                    /* pulse-group noise events */

/*
 * Signal/noise ratios. These reveal signal quality and health of the
 * tracking processes.
 */
double psnr = 0;                  /* pulse-group max-envelope/rms */
double esnr = 0;                  /* envelope rms-max/rms-min */

/*
 * Main program
 *
 * Programming note: There is usually enough time between gri intervals
 * for one print statement, but not two, at least on a 286.
 */
main(argc, argv) int argc; char *argv[]; {
     int mindex = 0;                   /* index of min cycle in envelope */
     int maxdex = 0;                   /* index of max cycle in envelope */
     int cycle = 0;               /* cycle counter */
     int count = 0;               /* utility counter */
     int icnt = 0;                /* integration cycle counter */
     int mcnt = 0;                /* median counter */
     int ecnt = 0;                /* envelope counter */
     int env = 0;                 /* envelope scan pointer */
     int envbot = 0;                   /* first cycle in envelope scan */
     int envtop = NENV-1;         /* last cycle in envelope scan */
     int i, j, temp;                   /* utility temps */
     char codesw = 0;             /* gri a/b switch */
     char pllsw = 0;                   /* enable pll switch */
     double dtemp, etemp, ftemp, gtemp; /* utility doubles */
     double fmax, fmin;           /* utility max/min values */
     char msg[80] = "\0";         /* status message */
     int tmp[MMAX];               /* int temporary list */
     double ftmp[MMAX];           /* double temporary list */

/*
 * Decode command-line arguments
 *
 * usage: <program name> <gri> <codes> <agc> <vco>
 * <gri>  assigned LORAN-C group repitition interval (default 9960)
 * <codes>    m for master, s for slaves (first one found) (default m)
 * <agc>  initial agc dac (0-255) (default AGC parameter)
 * <vco>  initial vco dac (0-255) (default VCO parameter)
 */
     if (argc > 1)
          sscanf(argv[1], "%i", &gri);
     if (argc > 2)
          pulse = *argv[2];
     if (argc > 3)
```

```
          sscanf(argv[3], "%lf", &agcdac);
     if (argc > 4)
          sscanf(argv[4], "%lf", &vcodac);
/*
 * Initialization
 *
 * This section runs only once. It resets the timing generator,
 * loads its registers with default values and clears arrays. The
 * program then simulates a "1" keystroke and sets the receiver gain
 * at minimum to begin receiver calibration.
 */
     outp(TGC, RESET); outp(TGC, LOAD+0x1f); /* reset STC chip */
     outp(TGC, LOADDP+MASTER); outp(TGD, 0xf0); outp(TGD, 0x8a);
     outp(TGC, LOADDP+1);
     for (i = 0; i < 5*3; i++) {
          outp(TGD, init[i]); outp(TGD, init[i]>>8);
          }
     outp(TGC, LOADARM+0x1f);     /* let the good times (mode 1) roll */
     sprintf(report, "Calibrating receiver");

     /*
      * Main loop
      *
      * This is the main receiver loop and runs until escaped by a ^C
      * signal. The main loop runs twice per frame or once each gri
      * (pulse groups a and b) and performs the main receiver update
      * between the end of pulse group b and the beginning of pulse group
      * a. While most program functions are completed in one frame, some
      * may persist indefinately until canceled by another keystroke or
      * automatically by the program.
      */
     while (1) {

          /*
           * Scan for keyboard functions
           *
           * This section tests for keyboard commands and decodes
           * keystrokes.
           */
          if (kbhit() != 0) {
               kbd = getch();
               switch (kbd) {

                    /*
                     * The following commands control the phase of the
                     * receiver frame relative to the received signal. These
                     * are normally needed only for manual signal
                     * acquisition.
                     */
                    case '+': /* shift frequency offset +step/gri */
                         freq += step; break;

                    case '-': /* shift frequency offset -step/gri */
                         freq -= step; break;
```

```
case '0': /* shift frequency to zero offset */
     freq = 0; break;

case ']': /* shift phase +10 us*step */
     phase = step; break;

case '[': /* shift phase -10 us*step */
     phase = -step; break;

/*
 * The following commands adjust various receiver vco
 * and agc bias values. The exact values are determined
 * at initial receiver alignment and compiled in the
 * source code and normally need not be changed in
 * regular operation.
 */
case '}': /* adjust program gain up */
     dervel *= 1.1; break;

case '{': /* adjust program gain down */
     dervel /= 1.1; break;

case ')': /* adjust receiver gain up */
     agcdac++; agc++; break;

case '(': /* adjust receiver gain down */
     agcdac--; agc--; break;

case '>': /* adjust vco bias up */
     vcodac++; break;

case '<': /* adjust vco bias down */
     vcodac--; break;

/*
 * The following commands select which pulse codes are
 * used and determine which set (a or b) to use. These
 * are normally needed only for manual signal
 * acquisition.
 */
case 'm': /* use master pulse codes */
     pulse = 'm'; break;

case 's': /* use slave pulse codes */
     pulse = 's'; break;

case 'x': /* flip pulse code a/b */
     codesw = !codesw; break;

/*
 * The following commands select the receiver gate and
 * integrator gain. These are normally needed only for
 * manual signal acquisition.
 */
case 'u': /* switch to ungated mode */
```

```
     par = 0x02; step = 10; pllsw = 0; break;

case ’g’: /* switch to gri mode */
     par = 0x06; step = 10; pllsw = 0; break;

case ’p’: /* switch to pci mode */
     par = 0x0a; step = 10; pllsw = 0; break;

case ’e’: /* switch to stb mode */
     par = 0x0c; step = 1; pllsw = 0; break;

case ’l’: /* open loop (for alignment) */
     pllsw = 1; break;

/*
 * The following commands establish the receiver mode.
 * Normally, the acquisition process sequences modes
 * automatically through four modes in the order below.
 * These commands can be used to restart the process at
 * any point.
 */
case ’1’: /* calibrate min gain */
     mode = kbd; par = 0x0a; break;

case ’2’: /* calibrate max gain */
     mode = kbd; par = 0x0a; break;

case ’3’: /* calibrate normal gain */
     mode = kbd; par = 0x0a; break;

case ’4’: /* search for pulse-group phase */
     mode = kbd; par = 0x0a; break;

case ’5’: /* search for envelope phase */
     mode = kbd; par = 0x0c; break;
/*
 * Display receiver status (debug). Note that the
 * display may take longer than a gri, so that the
 * receiver can loose synchronization. Usually,
 * synchronization can be resynchronized simply by
 * flipping the code phase ("x" command).
 */
case ’q’:
     outp(TGC, LOADDP+MASTER);
     printf("status %02x  master mode %02x %02x",
          inp(TGC), inp(TGD), inp(TGD));
     outp(TGC, LOADDP+1);
     printf("counters\n");
     for (i = 1; i < 6; i++) {
          printf("%2i", i);
          for (j=1; j<7; j++) printf(" %02x",
               inp(TGD));
          printf("\n");
          }
     printf("agcdac %4.1lf  vcodac %4.1lf\n",
```

```
                    agcdac, vcodac);
                break;
            }
        }

    /*
     * Wait for next gri and accumulate i, q, agc
     *
     * This section first enables automatic adc start at the end of
     * the next gri. It then reads the i integrator (adc channel 0),
     * q integrator (adc channel 1) and agc (adc channel 2). These
     * values are summed for the a and b pulse groups and processed
     * at the end of the b pulse group.
     */
    outp(PAR, ENG | par); outp(ADC, 0);        /* i */
    while ((inp(ADCGO)&DONE) == 0);
    isig += inp(ADC);
    outp(PAR, par);
    outp(ADC, 1); outp(ADCGO, START);     /* q */
    while ((inp(ADCGO)&DONE) == 0);
    qsig += inp(ADC);
    outp(ADC, 2); outp(ADCGO, START);     /* agc */
    while ((inp(ADCGO)&DONE) == 0);
    agcraw += inp(ADC);

    /*
     * Process i-phase, q-phase and agc
     *
     * Note that a LORAN frame consists of two gri intervals a and
     * b, each with individual pulse codes. The receiver integrates
     * each gri using the assigned pulse codes. There are two sets
     * of pulse codes, one for the master station and the other for
     * slave stations, of which there may be as many as four. Each
     * LORAN chain is assigned a unique gri interval in the range
     * 40-100 ms.
     */
    if (codesw == 0) {
        count++;

        /*
         * gri a processing
         *
         * This section processes the i, q and agc samples from the
         * previous frame and computes the receiver vco and agc dac
         * values. It begins by computing the average in-phase,
         * quadrature-phase and agc signals. Note the offset and
         * sign corrections, since the adc operates in unsigned,
         * inverted mode.
         */
        isig = -(isig-iofs)*gain; qsig = -(qsig-qofs)*gain;
        agcavg += (agcraw-agcofs)/AGCFAC;
        if (agcavg < 0) agcavg = 0;

        /*
         * In calibrate mode "1" the receiver gain is set to
```

```
 * minimum. The program waits for the averages to settle
 * and then calculates the integrator offsets and agc
 * minimum value, which takes a few seconds. When done, the
 * program simulates a ’2’ keystroke to complete receiver
 * calibration.
 *
 * In this mode the vco dac is clamped and the agc dac is
 * set to minimum (0).
 */
if (mode == ’1’) {
     vco = vcodac; agc = 0; env = 0;
     iofs -= isig/AGCFAC; qofs -= qsig/AGCFAC;
     dtemp = agcraw-agcofs; agcofs += dtemp/AGCFAC;
     if (isig*isig+qsig*qsig+dtemp*dtemp < 3 &&
          count > AGCFAC*2) {
          mode = ’2’;        /* continue in mode ’2’ */
          count = 0;
          sprintf(report, "iofs %4.1lf  qofs %4.1lf  agcofs %4.1lf  gain %4.
               iofs, qofs, agcofs, gain);
          }
     }

/*
 * In calibrate mode 2 the receiver gain is set to  maximum.
 * The program waits for the averages to settle and then
 * calculates the agc maximum value, which takes a few
 * seconds. When done, the program simulates a ’3’ keystroke
 * to establish the normal receiver gain configuration.
 *
 * In this mode the vco dac is clamped and the agc dac is
 * set to maximum (255).
 */
else if (mode == ’2’) {
     vco = vcodac; agc = 255;
     dtemp = agcraw-agcmax; agcmax += dtemp/AGCFAC;
     if (dtemp*dtemp < 1 && count > AGCFAC*2) {
          mode = ’3’;        /* continue in mode 3 */
          count = 0; agcavg = 0;
          }
     }

/*
 * In calibrate mode 3 the receiver gain is determined by
 * the receiver agc, which is a peak-indicating type
 * sensitive to the peak pulse power in the 90-110 kHz
 * spectrum. This step, which takes a few seconds, is
 * necessary only to insure the receiver operates at the
 * best signal/noise ratio and without overload during the
 * scanning process. When done, the program simulates a ’4’
 * keystroke to begin the pulse-group scan.
 *
 * In this mode the vco dac is clamped and the agc dac is
 * computed from the receiver agc smoothed output.
 *
 * *** This part not finished yet ***
```

```
 */
else if (mode == '3') {
     vco = vcodac; agc = agcdac;
     if (count > AGCFAC*2) {
          mode = '4';         /* continue in mode 4 */
          count = 0; fmax = 0; fmin = 0; pgcnt++;
          sprintf(report, "agcmax %4.1lf  agcavg %4.1lf\nSearching for signa
               agcmax, agcavg);
          }
     }

/*
 * In acquisition mode 4 the radio scans at 100 us per frame
 * over the entire frame (2*gri), which takes up to about
 * seven minutes. The program accumulates the 12 most recent
 * received rms signal samples in a pulse-gate filter. The
 * first two samples are used as a noise gate. In the three
 * most recent samples, including two in the filter and the
 * most recent, either the first or last must be at least
 * 1/3 the second, or the second is most likely a noise
 * pulse, which could be due to a pulse code from another
 * chain just happening to appear in the pulse-code window.
 * In addition, the program computes a weighted sum which
 * favors the two samples near the middle of the last ten
 * stages in the filter. The program saves the value and
 * index of the highest sample received. Note that sample
 * selection from the pulse-group filter is valid only after
 * 12 samples have been received. Therefore, the program
 * waits for 12 samples before updating a bin and wraps
 * around for 12 bins beyond the end of the frame. In order
 * to improve the estimated position, the program
 * interpolates between the intervals just before and just
 * after the selected interval. When the scan is complete,
 * the program simulates a '5' keystroke to begin the
 * envelope scan.
 *
 * In this mode the vco dac is clamped and the agc dac
 * remains at the value computed in the previous mode.
 */
else if (mode == '4') {
     vco = vcodac; agc = agcdac;
     if (count > 2*gri/10+NRMS) {
          fmin = sqrt(fmin/count);
          if (fmax <= 4*fmin) {
               status(fmax, fmin, " low signal");
               count = 0; fmax = 0; fmin = 0; pgcnt++;
               }
          else {
               status(fmax, fmin, " cycle search");
               if (fmin > 0)
                    psnr = fmax/fmin;
               count = 0; pllsw = 0;
               phase = mindex-offset-OFFSET; freq = 0;
               mode = '5';         /* continue in mode 5 */
               envbot = 0; envtop = NENV-1;
```

```
                env = 0; ptrenv = 0; par = 0x0c;
                for (i = 0; i < MMAX; i++)
                     stb[i] = 0;
                efac = EMIN; pfac = PMIN;
                esnr = 0; mcnt = 0; ecnt = 0;
                strobe = 0; fmin = 999; fmax = 0;
                icnt = 0; encnt++;
                }
           }
      else {
           freq = 10;
           dtemp = isig*isig+qsig*qsig;
           fmin += dtemp;
           dtemp = sqrt(dtemp);
           if (rms[0] > dtemp*PGATE ||
                rms[0] > rms[1]*PGATE) {
                rms[0] = (dtemp+rms[1])/2; pncnt++;
                }
           for (i = 0; i < NRMS-1; i++) {
                rms[i+1] = rms[i];
                }
           rms[0] = dtemp;
           dtemp = rms[4]+rms[5]+rms[6];
           dtemp -= (rms[1]+rms[2]+rms[3]+
                rms[7]+rms[8]+rms[9])/6;
           if (count > NRMS) {
                if (dtemp > fmax) {
                     fmax = dtemp;
                     dtemp = (rms[6]-rms[4])/
                          (rms[6]+rms[4]);
                     mindex = offset+(int)dtemp*10;
                     }
                }
           if (((count+1)%100) == 0)
                status(fmax, sqrt(fmin/count), "\0");
           }
      }

/*
 * In tracking mode 5, which is the default, the program
 * scans at 10 us per frame over the 300-us receiver
 * pulse-gate interval. The program integrates the q-
 * channel and i-channel envelopes separately in one-cycle
 * (10 us) bins and determines the reference cycle as the
 * minimum rms error relative to a prestored model envelope.
 * If sufficient integrated signal amplitude is not found
 * within a minute or so, the search is abandoned and the
 * program reverts to the pulse-group search.
 *
 * In this mode the vco dac follows the i channel and the
 * agc follows the q channel, both averaged in complicated
 * ways.
 */
else {
```

```
/*
 * This is the envelope scan. The q signal represents
 * the amplitude and the i signal the phase-correction.
 * The program actually uses only the envelope
 * amplitude.
 */
ftemp = sqrt(isig*isig+qsig*qsig);
gtemp = isig;

/*
 * Median filters for both the amplitude and phase
 * signals for each cycle of the pulse are used to help
 * cope with cross-rate interference and dual-rate
 * transmitter blanking. The variables pmed and amed are
 * shift registers containing the most recent samples.
 * The variables ftemp and gtemp are the median
 * amplitude and phase channels, respectively, while the
 * variable mspan is the span of the envelope channel
 * for later use as a noise gate.
 */
if (mcnt < MMIN)
     mcnt++;
for (i = 0; i < mcnt-1; i++) {
     pmed[env][i+1] = pmed[env][i];
     emed[env][i+1] = emed[env][i];
     }
pmed[env][0] = ftemp;
emed[env][0] = gtemp;
for (i = 0; i < mcnt; i++){
     ftmp[i] = pmed[env][i];
     for (j = 0; j < i; j++) {
          if (ftmp[i] < ftmp[j]) {
               gtemp = ftmp[i]; ftmp[i] = ftmp[j];
               ftmp[j] = gtemp;
               }
          }
     }
ftemp = ftmp[mcnt/2];
mgate = ftmp[mcnt-1]-ftmp[0];
for (i = 0; i < mcnt; i++){
     ftmp[i] = emed[env][i];
     for (j = 0; j < i; j++) {
          if (ftmp[i] < ftmp[j]) {
               gtemp = ftmp[i]; ftmp[i] = ftmp[j];
               ftmp[j] = gtemp;
               }
          }
     }
gtemp = ftmp[mcnt/2];
ecyc[env] += (ftemp-ecyc[env])/efac;
pcyc[env] += (gtemp-pcyc[env])/pfac;

/*
 * Experiment: compute derived envelope.
 */
```

```
          if (env == envtop)
               ftemp = ecyc[env];
          else
               ftemp = ecyc[env+1];
          edrv[env] = ecyc[env]-dervel*(ftemp-ecyc[env]);
          if (edrv[env] < 0)
               edrv[env] = -edrv[env];

          /*
           * Compute rms envelope error and find cycles of max
           * amplitude and min error.
           */
          if (ecyc[env] > fmax) {
               fmax = ecyc[env]; maxdex = env;
               }
          if (env+6 > NENV-1)
               ftemp = ecyc[NENV-1];
          else
               ftemp = ecyc[env+6];
          if (ftemp != 0)
               ftemp = 100/ftemp;
          etemp = 0;
          for (i = 0; i < 7; i++) {
               j = env+i;
               if (j > envtop)
                    dtemp = ecyc[envtop]*ftemp;
               else
                    dtemp = ecyc[j]*ftemp;
               dtemp -= envcyc[i]; etemp += dtemp*dtemp;
               }
          erms[env] = sqrt(etemp/7);
          if (erms[env] < fmin) {
               fmin = erms[env]; mindex = env;
               }

/*        if (edrv[env] < fmin &&
               env < maxdex-1 && env > maxdex-7) {
               fmin = edrv[env]; mindex = env-2;
               }
*/

          /*
           * This section is entered at the end of the envelope
           * scan. It establishes the cycle position and
           * calculates the signal/noise ratio and phase-
           * correction signal and receiver gain control. The
           * cycle position is determined from the envelope cycle
           * of minimum rms error using a median filter. The
           * phase-correction signal is extracted from the third
           * carrier cycle, while the signal/noise ratio is
           * computed as the ratio of the amplitude of the seventh
           * carrier cycle divided by the rms error of the first
           * cycle. The span of the values in the strobe filter is
           * calculated for later use as a noise gate. In order to
           * maintain critical damping (damping factor of 0.707),
```

```
 * the pll gain must be controlled so that the product
 * of the loop time constant and loop gain is equal to
 * 1/2.
 */
if (env == envbot) {
     fmin = 999; fmax = 0;
     for (i = 0; i < mcnt-1; i++)
          stb[i+1] = stb[i];
     if (maxdex < 8)
          stb[0] = 0;
     else if (mindex < maxdex-8 || mindex > maxdex-4)
          stb[0] = maxdex-6;
     else
          stb[0] = mindex;
     for (i = 0; i < mcnt; i++){
          tmp[i] = stb[i];
          for (j = 0; j < i; j++) {
               if (tmp[i] < tmp[j]) {
                    temp = tmp[i]; tmp[i] = tmp[j];
                    tmp[j] = temp;
                    }
               }
          }
     cycle = tmp[mcnt/2];
     ftemp = 1/(2*VGAIN*pfac);
     if (ftemp > 1)
          ftemp = 1;
     ftemp *= pcyc[cycle+2];
     if (pllsw != 0) ftemp = 0;
     vco = vcodac+ftemp;
     if (cycle < 1 || cycle > 17)
          sgate = SGATE;
     else
          sgate = tmp[mcnt-1]-tmp[0];
     dtemp = ecyc[cycle+6]; etemp = erms[cycle];
     if (etemp > 0)
          esnr = dtemp/etemp;
     else
          esnr = 0;
     strcpy(msg, "\0");

     /*
      * This section is entered at the end of each
      * integration cycle. It checks the signal quality,
      * cycle position within the 300-us envelope gate
      * and strobe position within the 90-us cycle-
      * identification window. The receiver gain is
      * determined from the amplitude of the seventh
      * cycle in the window. If reliable cycle
      * identification has not been achieved in a
      * reasonable time, the program punts back to the
      * pulse-group scan. The integration time constant
      * is increased only if all samples in the strobe
      * median filter are identical.
      */
```

```
icnt++;
if (icnt >= mcnt) {
      icnt = 0;
      if (mcnt < (int)(8*pfac) && mcnt < MMAX)
            mcnt++;
      if (sgate >= SGATE) {
            if (count < WATCHDOG) {
                  strcpy(msg, " strobe noise");
                  if (strobe != 0) {
                        mode = '5';    /* continue in mode 5 */
                        envbot = 0; envtop = NENV-1;
                        env = 0; ptrenv = 0; par = 0x0c;
                        for (i = 0; i < MMAX; i++)
                              stb[i] = 0;
                        efac = EMIN; pfac = PMIN;
                        esnr = 0; mcnt = 0; ecnt = 0;
                        strobe = 0; fmin = 999; fmax = 0;
                        encnt++;
                        }
                  }
            else {
                  strcpy (msg,
                        " resume pulse-group search");
                  mode = '4';    /* continue in mode 4 */
                  count = 0; fmax = 0; fmin = 0; pgcnt++;
                  par = 0x0a;
                  }
            }
      else {
            if (cycle < 3 || cycle >= 15) {
                  strcpy(msg, " cycle slip");
                  phase = -(9-cycle); cscnt++;
                  mode = '5';    /* continue in mode 5 */
                  envbot = 0; envtop = NENV-1;
                  env = 0; ptrenv = 0; par = 0x0c;
                  for (i = 0; i < MMAX; i++)
                        stb[i] = 0;
                  efac = EMIN; pfac = PMIN;
                  esnr = 0; mcnt = 0; ecnt = 0;
                  strobe = 0; fmin = 999; fmax = 0;
                  cscnt++;
            }
            else if (cycle+2 != strobe) {
                  strcpy(msg, " strobe slip");
                  mode = '6'; sscnt++;
                  strobe = cycle+2;
                  }
            }
      }
ecnt++;
if (ecnt >= (int)efac) {
      ecnt = 0;
      if (strobe != 0) {
            if (dtemp > 100)
                  agc--;
```

```
          else
               agc++;
          }
     if (sgate == 0) {
          if (efac < EMAX)
               efac *= EFAC;
          if (pfac < PMAX)
               pfac *= PFAC;
          }
     }

/*
 * This section determines what to display about the
 * status of the receiver and tracking information.
 * The envelope amplitude phase-correction and rms
 * error signals can be desplayed for every cycle
 * scanned. In coarse-search mode before the strobe
 * position has been determined only the
 * signal/noise ratio and strobe span are displayed;
 * while, in fine-tracking mode only the three
 * cycles before and five cycles after the strobe
 * are displayed. This includes one cycle before the
 * cycle-posigion window and one after. This reduces
 * the integrator bumps in case of a strobe slip,
 * which normally is not more than +-1 cycle.
 */
if (strobe == 0) {
     strcat(msg, " ?");
     dtemp = esnr; etemp = sgate;
     ptrenv = cycle+2;
     }
else {
     count = 0;
     dtemp = ecyc[ptrenv]; etemp = erms[ptrenv];
     if (ptrenv == strobe)
               strcat(msg, " x");
     if (kbd == ’v’)
          etemp = pcyc[ptrenv];
     if (ptrenv == envtop-1) {
          strcat(msg, " agc");
          etemp = agc;
          }
     else if (ptrenv == envtop) {
          strcat(msg, " vco");
          etemp = vco;
          }
     }
status(dtemp, etemp, msg);
if (strobe != 0) {
     envbot = strobe-3; envtop = strobe+5;
     }
else {
     envbot = 0; envtop = NENV-1;
     }
ptrenv++;
```

```
          if (ptrenv > envtop) ptrenv = envbot;
          if (ptrenv < envbot) ptrenv = envtop;
          }
     }

/*
 * Compute the interval to the next gri. Normally, this is
 * fixed at the gri interval specified less 800 (8 ms), but
 * can be adjusted in 10-us steps to align the timing
 * generator to the transmitted LORAN-C signal. The
 * adjustments can be in the form of a frequency offset, in
 * 10 us/frame increments, or a one-time phase adjustment,
 * in 10-us increments. Note that the interval to the next
 * gri has already been loaded in the counters at this
 * point. The value loaded here actually applies to the gri
 * after that. Since this is the end of a b interval and the
 * next a interval has already been established, frequency
 * and phase adjustments will take effect at the beginning
 * of the next b interval after that, so correct timing
 * between the a and b pulse groups for a single frame are
 * preserved. Also, if a phase adjustment occurs,purge the
 * envelope averages.
 */
offset += freq+phase;        /* adjust epoch */
while (offset >= 2*gri) {
     offset -= 2*gri; frame++;
     }
while (offset < 0) {
     offset += 2*gri; frame--;
     }
temp = gri+freq+phase;       /* gri counter */
while (temp >= 2*gri)
     temp -= 2*gri;
while (temp < FGUARD || (temp < DGUARD &&
     report[0] != '\0'))
     temp += 2*gri;
temp -= 800; outp(TGC, LOADDP+0x0a);
outp(TGD, temp); outp(TGD, temp>>8); phase = 0;

/*
 * Load the vco and agc dacs and initialize the pulse code
 * and internal integrators for the next frame. Also step
 * the envelope strobe one cycle (10 us) for the next frame.
 */
dtemp = vco;                 /* vco dac */
if (dtemp > 255)
     dtemp = 255;
if (dtemp < 0)
     dtemp = 0;
outp(DACA, (int)dtemp);
dtemp = agc;                 /* agc dac */
if (dtemp > 255)
     dtemp = 255;
if (dtemp < 0)
     dtemp = 0;
```

```
            outp(DACB, (int)dtemp);
            temp = 5000-pcx;             /* envelope scan window */
            outp(TGC, LOADDP+0x0b);
            outp(TGD, pcx); outp(TGD, pcx>>8);
            outp(TGD, temp); outp(TGD, temp>>8);
            outp(TGC, LOADDP+0x0c);
            env--;                            /* envelope scan strobe */
            if (env < envbot)
                 env = envtop;
            if (env > envtop)
                 env = envbot;
            temp = env*50+RCVDELAY;
            outp(TGD, temp); outp(TGD, temp>>8);
            if (pulse == 's')
                 outp(CODE, SPCA); /* pulse code */
            else
                 outp(CODE, MPCA);
            isig = 0; qsig = 0; agcraw = 0;
            }
       else {

            /*
             * gri b processing
             *
             * This section sets up for the b pulse group. It resets the
             * gri (counter 2) load register to delay exactly one gri
             * less 800 (8 ms), which will be used for the subsequent a
             * interval following the next b interval. It also sets the
             * b code for the next b interval. The program also displays
             * the report string if left by the preceeding a interval.
             */
            temp = gri-800;                   /* gri counter */
            outp(TGC, LOADDP+0x0a);
            outp(TGD, temp); outp(TGD, temp>>8);
            if (pulse == 's')
                 outp(CODE, SPCB);
            else
                 outp(CODE, MPCB);
            if (report[0] != '\0') {
                 puts(report); report[0] = '\0';
                 }
            }
       codesw = !codesw;
       }
  }

/*
 * subroutine to encode and display status line
 * displays line of the form
 *
 * kggggm n ttt ff ooooo cc uu.u vv.v message
 *
 *  kbd   assigned gri
 *  k          last keystroke
 *  gggg  assigned gri
```

```
 *   m          master (m) or slave (s) indicator
 *   n          operating mode number
 *   ttt        time constant
 *   ff         frame offset relative to turnon
 *   ooooo      cycle offset within the frame (10 us steps) to gri
 *   cc         cycle offset within the gri (10 us steps)
 *   uu.u signal value at this position
 *   vv.v error value at this position
 *   message    information message
 */
void status(val1, val2, string) double val1, val2; char *string; {

     if (kbd == 'r') {
          sprintf(report, "pg%3i en%3i cs%3i ss%3i pn%5i vc%5.1lf es%5.1lf gn%5.2lf",
               pgcnt, encnt, cscnt, sscnt, pncnt, psnr, esnr,
               gain);
          kbd = ' ';
          }
     else
          sprintf(report, "%c%5i%c %c%6.1f%3i%6i%3i%6.1lf%6.1lf%6.1lf%s",
               kbd, gri, pulse, mode, pfac, frame, offset+strobe, ptrenv,
               val1, edrv[ptrenv], val2, string);
     }

/* end program */
```