```
// TollFile.cpp - TOLL GATE SYSTEM (RECORD VEHICLE ID'S IN A FILE)
//
// MODULE INDEX
// NAME                         CONTENTS
// ErrSys                       Process system error
// ErrApp                       Process application error
// AlarmHandler                 Alarm signal handler
// ClkTickGen                   Clock tick generator
// ReplyRcvr                    Reply receiver process
// SendPoll                     Send a poll to the sensor network
// StateMach                    State machine
// main                         Main line
//
// MAINTENANCE HISTORY
// DATE         PROGRAMMER AND DETAILS
// 02-10-04     JS      Original
//
//-----------------------------------------------------------------------------

#include <sys/types.h>          // System type declarations
#include <string.h>             // String manipulation functions
#include <signal.h>             // Signal codes
#include <unistd.h>             // Operating system standard functions
#include <termios.h>            // Serial port parameter structure
#include <fcntl.h>              // File control options
#include <errno.h>              // Operating system error codes
#include <time.h>               // Time declarations
#include <sys/time.h>           // Interval timer declarations
#include <iostream>             // C++ input/output streams
#include <iomanip>              // C++ input/output manipulators
#include <fstream>              // File stream declarations
#include <list>                 // C++ lists
using namespace std;            // Expand standard namespace to global scope

//-----------------------------------------------------------------------------

// DEFINITIONS

const size_t            VEHICLE_ID_LEN = 16;
                                // Vehicle identifier length
const size_t            MAX_SENSOR_MSG_LEN = 32;
                                // Maximum sensor message length
const unsigned long     CLK_TICK_FREQ = 20;
                                // Clock tick frequency
const unsigned long     FLUSH_TIME = CLK_TICK_FREQ * 3 / 10;
                                // Error flushing time
const unsigned long     REPLY_TIMEOUT_TIME = CLK_TICK_FREQ * 3 / 10;
                                // Reply timeout time
const static char       SENSOR_PORT_NAME[] = "/dev/tty1a";
                                // Sensor port name
const static unsigned char SENSOR_ADR_ARR[] = {0x04, 0x05, 0x08, 0x09};
                                // Sensor address array
const size_t            SENSOR_ADR_CNT = sizeof(SENSOR_ADR_ARR)
                                / sizeof(SENSOR_ADR_ARR[0]);
                                // Sensor address count
const unsigned char     POLLER_ADR = 0x00;
                                // Address of the poller on the sensor
                                // network
const static char       TRANS_FILE_NAME[] = "TRANSFILE";
                                // Transaciton file name
const time_t            TRANS_EXPIRY_TIME = 30;
                                // Transaction expiry time

//-----------------------------------------------------------------------------

// SENSOR MESSAGE TYPES

const unsigned char     SENSOR_MT_POLL = 0x01;
                                // Poll command
const unsigned char     SENSOR_MT_VEHICLE_DETECTED = 0x02;
                                // Vehicle detected reply
const unsigned char     SENSOR_MT_NO_VEHICLE = 0x03;
                                // No vehicle reply

//-----------------------------------------------------------------------------

// VEHICLE IDENTIFIER TYPE

typedef unsigned char VehicleId_t[VEHICLE_ID_LEN];

//-----------------------------------------------------------------------------
```

```
// SENSOR REPLY FAULT EXCEPTION

class ReplyFault_c {
public:
    const char          *replyFaultMsg;         // Reply fault message
    ReplyFault_c (const char *msg) : replyFaultMsg (msg) {}
                                                // Constructor
};

//----------------------------------------------------------------------------

// INTERNAL STIMULUS TYPE CODES

enum StimTypeCode_t {
    STIM_TICK,                  // Clock tick
    STIM_REPLY_DATA             // Reply data
};

//----------------------------------------------------------------------------

// INTERNAL MESSAGE STRUCTURE

struct Stimulus_t {
    StimTypeCode_t      stimTypeCode;           // Message type code
    size_t              stimByteCnt;            // Byte count
    unsigned char       stimByteArr[MAX_SENSOR_MSG_LEN]; // Byte array
};

//----------------------------------------------------------------------------

// PENDING TRANSACTION STRUCTURE

struct PendTrans_t {
    VehicleId_t         pendTransVehicleId;     // Vehicle identifier
    time_t              pendTransLastReplyTime; // Time of last reply to poll
};

//----------------------------------------------------------------------------

// GLOBAL VARIABLES

int             stateMachQueWrFd;       // State Mach Queue write file descr

//----------------------------------------------------------------------------

// PROCESS SYSTEM ERROR

void
ErrSys (
    const char          *msg)           // Error message
{
    int                 savedErrNo;     // Saved error number

    // Report the error

    savedErrNo = errno;
    cerr << "Error: " << msg << ": " << strerror (savedErrNo) << '\n';

    // Terminate all the processes

    kill (0, SIGTERM);
    exit (1); // Just to be safe
}

//----------------------------------------------------------------------------

// PROCESS APPLICATION ERROR

void
ErrApp (
    const char          *msg)           // Error message
{
    // Report the error

    cerr << "Error: " << msg << '\n';

    // Terminate all the processes

    kill (0, SIGTERM);
    exit (1); // Just to be safe
}
```

```
//---------------------------------------------------------------------------

// ALARM SIGNAL HANDLER

void
AlarmHandler (
    int                 sig)            // Signal number
{
    Stimulus_t          stimulus;       // Stimulus structure

    // Validate the received signal

    if (sig != SIGALRM)
        ErrApp ("unexpected signal");

    // Send a clock tick stimulus to the State Machine

    stimulus.stimTypeCode = STIM_TICK;
    if (write (stateMachQueWrFd, &stimulus, sizeof(stimulus)) !=
      static_cast<ssize_t>(sizeof(stimulus)))
        ErrSys ("write to pipe");
}

//---------------------------------------------------------------------------

// CLOCK TICK GENERATOR

void
ClkTickGen ()
{
    struct itimerval    iTimVal;        // Interval timer value
    struct sigaction    alarmSa;        // Alarm signal action

    // Vector alarm signals to the Alarm Signal Handler

    memset (&alarmSa, 0, sizeof(alarmSa));
    alarmSa.sa_handler = AlarmHandler;
    sigemptyset (&alarmSa.sa_mask);
    sigaddset (&alarmSa.sa_mask, SIGALRM);
    if (sigaction (SIGALRM, &alarmSa, 0) == -1)
        ErrSys ("sigaction SIGALRM");

    // Configure the interval timer to generate an alarm every
    // clock tick.

    memset (&iTimVal, 0, sizeof(iTimVal));
    iTimVal.it_interval.tv_sec = 0;
    iTimVal.it_interval.tv_usec = 1000000 / CLK_TICK_FREQ;
    iTimVal.it_value = iTimVal.it_interval;
    if (setitimer (ITIMER_REAL, &iTimVal, 0) == -1)
        ErrSys ("setitimer");

    // Pause until terminated by a signal

    for (;;) pause ();
}

//---------------------------------------------------------------------------

// REPLY RECEIVER PROCESS

void
ReplyRcvr (
    int                 sensorFd)       // Sensor port file descriptor
{
    Stimulus_t          stimulus;       // Stimulus structure
    ssize_t             rdLen;          // Read length

    // Loop receiving data from the sensor port

    for (;;) {

        // Receive some data

        errno = 0;
        rdLen = read (sensorFd, stimulus.stimByteArr, MAX_SENSOR_MSG_LEN);
        if (rdLen <= 0 || rdLen > MAX_SENSOR_MSG_LEN)
            ErrSys ("read reply data");

        // Send the reply data to the State Machine

        stimulus.stimTypeCode = STIM_REPLY_DATA;
```

```
        stimulus.stimByteCnt = rdLen;
        if (write (stateMachQueWrFd, &stimulus, sizeof(stimulus)) !=
          static_cast<ssize_t>(sizeof(stimulus)))
            ErrSys ("write to pipe");
    }
}

//----------------------------------------------------------------------------

// SEND A POLL TO THE SENSOR NETWORK

void
SendPoll (
    int                 sensorFd,       // Sensor port file descriptor
    size_t              sensorInd)      // Sensor index
{
    unsigned char       pollBuf[MAX_SENSOR_MSG_LEN]; // Poll buffer
    size_t              pollLen;        // Poll message length
    unsigned char       lrc;            // Longitudinal redundancy check
    int                 i;              // General purpose index

    // Construct the poll message

    pollLen = 0;
    pollBuf[pollLen++] = SENSOR_ADR_ARR[sensorInd];
    pollBuf[pollLen++] = POLLER_ADR;
    pollBuf[pollLen++] = SENSOR_MT_POLL;
    pollBuf[pollLen++] = 0; // Payload length
    lrc = 0;
    for (i = 0; i < pollLen; i++) lrc ^= pollBuf[i];
    pollBuf[pollLen++] = lrc;

    // Send the poll message

    if (write (sensorFd, pollBuf, pollLen) != pollLen)
        ErrSys ("write to sensor port");
}

//----------------------------------------------------------------------------

// STATE MACHINE

void
StateMach (
    int                 sensorFd,       // Sensor port file descriptor
    int                 stateMachQueRdFd) // State Mach Queue read file descr
{
    list<PendTrans_t>   pendList;       // Pending transaction list
    list<PendTrans_t>::iterator nextPend; // Next pending transaction
    list<PendTrans_t>::iterator thisPend; // This pending transaction
    PendTrans_t         pendTrans;      // Pending transaction structure
    Stimulus_t          stimulus;       // Stimulus structure
    bool                fetchPending;   // Fetch pending flag
    bool                flushing;       // Flushing serial data after an error
    long                ticksRem;       // Clock ticks remaining
    long                ticksSinceRecv; // Clock ticks since last received char
    time_t              msgTime;        // Message time
    time_t              expTime;        // Message expiry time
    size_t              sensorInd;      // Sensor index
    size_t              replyLen;       // Reply length
    unsigned char       replyBuf[MAX_SENSOR_MSG_LEN]; // Reply buffer
    unsigned char       lrc;            // Longitudinal redundancy check
    const unsigned char *vehicleId;     // Pointer to the vehicle identifier
    int                 i;              // General purpose index
    ofstream            transFile;      // Transaction file

    // Open the transaction file

    transFile.open (TRANS_FILE_NAME, ios_base::app);
    if ( ! transFile) ErrSys ("open transaction file");
    transFile << hex << setfill('0');

    // Initialise the process state

    fetchPending = 0;
    flushing = 1;
    ticksRem = FLUSH_TIME;
    sensorInd = 0;
    ticksSinceRecv;

    // Process stimuli received in the State Machine Queue
```

```
for (;;) {

    // Receive a stimulus

    if (read (stateMachQueRdFd, &stimulus, sizeof(stimulus))
      != static_cast<ssize_t>(sizeof(stimulus)))
        ErrSys ("read stimulus");
    msgTime = time(0);
    expTime = msgTime - TRANS_EXPIRY_TIME;

    // Catch sensor reply fault exceptions

    try {
        // Process the stimulus according to its type

        switch (stimulus.stimTypeCode) {

        // Process a clock tick

        case STIM_TICK:

            // Increment the number of ticks since data was last received.

            ticksSinceRecv ++ ;

            // Decrement the number of ticks remaining.  If more ticks
            // remain, wait for another tick.

            ticksRem -- ;
            if (ticksRem > 0) break;

            // If the State Machine is currently flushing received
            // characters after an error and the flush time has expired,
            // send the next poll.

            if (flushing) {
                flushing = 0;
                replyLen = 0;
                ticksRem = REPLY_TIMEOUT_TIME;
                SendPoll (sensorFd, sensorInd);
            }

            // If the State Machine is receiving a reply and the timeout
            // has expired, throw a timeout fault.

            else
                throw ReplyFault_c ("timeout");
            break;

        // Process reply data

        case STIM_REPLY_DATA:

            // Reset the number of ticks since data was last received.

            ticksSinceRecv = 0;

            // If the State Machine is currently flushing received
            // characters after an error reset the number of ticks
            // remaining.

            if (flushing) {
                ticksRem = FLUSH_TIME;
            }

            // If the State Machine is receiving a reply, append the
            // received data to the reply buffer and process the
            // reply.

            else {

                // Check that the maximum message length has not
                // been exceeded.

                if (replyLen + stimulus.stimByteCnt > MAX_SENSOR_MSG_LEN)
                    throw ReplyFault_c ("reply data overrun");

                // Append the received data to the reply reply buffer

                memcpy (replyBuf + replyLen, stimulus.stimByteArr,
                    stimulus.stimByteCnt);
                replyLen += stimulus.stimByteCnt;
```

```
// Validate the reply body length

if (replyLen >= 4 && replyBuf[3] > MAX_SENSOR_MSG_LEN-5)
    throw ReplyFault_c ("reply body too long");

// If the full reply has been received, validate and
// process it.

if (replyLen >= 4 && replyLen >= replyBuf[3]+5) {

    // Check for extra data

    if (replyLen >= 4 && replyLen > replyBuf[3]+5)
        throw ReplyFault_c ("extra data");

    // Validate the LRC

    lrc = 0;
    for (i = 0; i < replyLen; i++) lrc ^= replyBuf[i];
    if (lrc != 0)
        throw ReplyFault_c ("LRC error");

    // Validate the destination address of the reply.

    if (replyBuf[0] != POLLER_ADR)
        throw ReplyFault_c ("wrong dest address");

    // Validate the source address of the reply.

    if (replyBuf[1] != SENSOR_ADR_ARR[sensorInd])
        throw ReplyFault_c ("wrong source address");

    // Process Vehicle Detected Reply

    if (replyBuf[2] == SENSOR_MT_VEHICLE_DETECTED) {

        // Validate the length of the vehicle identifier

        if (replyBuf[3] < VEHICLE_ID_LEN)
            throw ReplyFault_c ("reply too short");

        // Load the vehicle identifier pointer

        vehicleId = replyBuf + 4;

        // Purge any expired entries from the pending
        // transactions list while checking for duplicate
        // replies.

        nextPend = pendList.begin();
        while (
            nextPend != pendList.end() && (
                nextPend->pendTransLastReplyTime<=expTime ||
                memcmp (nextPend->pendTransVehicleId,
                    vehicleId, VEHICLE_ID_LEN) != 0
            )
        ) {
            thisPend = nextPend;
            nextPend ++ ;
            if (thisPend->pendTransLastReplyTime <= expTime)
                pendList.erase (thisPend);
        }

        // If the reply is a duplicate, update the last
        // poll time of the transaction.

        if (nextPend != pendList.end()) {
            nextPend->pendTransLastReplyTime = msgTime;
        }

        // If the reply is not a duplicate, append the
        // transaction to the pending transactions list
        // and the Transaction File.

        else {

            // Append the pending transaction to the
            // internal list.

            memcpy (pendTrans.pendTransVehicleId,
                vehicleId, VEHICLE_ID_LEN);
```

```
                                pendTrans.pendTransLastReplyTime = msgTime;
                                pendList.insert (pendList.end(), pendTrans);

                                // Append the transaction to the Transaction
                                // File.

                                for (i = 0; i < VEHICLE_ID_LEN; i++)
                                    transFile << setw(2) << int(vehicleId[i]);
                                transFile << '\n' << flush;
                            }
                        }

                        // Process No-Vehicle Reply

                        else if (replyBuf[2] == SENSOR_MT_NO_VEHICLE) {
                            // Empty
                        }

                        // Other replies are unacceptable

                        else
                            throw ReplyFault_c ("unexpected reply");

                        // Initiate a poll of the next sensor

                        sensorInd = (sensorInd + 1) % SENSOR_ADR_CNT;
                        replyLen = 0;
                        ticksRem = REPLY_TIMEOUT_TIME;
                        SendPoll (sensorFd, sensorInd);
                    }
                }
                break;

            // Other stimulus types are unacceptable

            default:
                ErrApp ("bad stimulus type code");
                // NOTREACHED
            }
        }

        // Process sensor reply fault exceptions

        catch (ReplyFault_c replyFault) {

            // Log a diagnostic message

            cerr << "Warning: " << replyFault.replyFaultMsg << " at sensor "
                << int(SENSOR_ADR_ARR[sensorInd]) << '\n';

            // Calculate the ticks remaining in the flushing period

            ticksRem = FLUSH_TIME - ticksSinceRecv;

            // If the flushing period has not yet expired, initiate
            // a flush.  Otherwise, initiate the next poll.

            sensorInd = (sensorInd + 1) % SENSOR_ADR_CNT;
            if (ticksRem > 0) {
                flushing = 1;
            } else {
                flushing = 0;
                replyLen = 0;
                ticksRem = REPLY_TIMEOUT_TIME;
                SendPoll (sensorFd, sensorInd);
            }
        }
    }
}

//-----------------------------------------------------------------------------

// MAIN LINE

int
main ()
{
    int                 sensorFd;       // Sensor port file descriptor
    termios             sensorTermios;  // Sensor port parameters
    int                 stateMachQue[2]; // State Mach Queue file descrs
    pid_t               clkTickGenPid;  // Clock Tick Generator proc id
    pid_t               replyRcvrPid;   // Reply Receiver proc id
```

```
    // Open the sensor port

    if ((sensorFd = open (SENSOR_PORT_NAME, O_RDWR)) == -1)
        ErrSys ("open serial port");

    // Configure the sensor port

    if (tcgetattr (sensorFd, &sensorTermios) == -1)
        ErrSys ("tcgetattr");
    sensorTermios.c_iflag = INPCK;
    sensorTermios.c_oflag = 0;
    sensorTermios.c_cflag = B9600 | CS8 | CREAD | PARENB | PARODD | CLOCAL;
    sensorTermios.c_lflag = 0;
    sensorTermios.c_cc[VMIN] = 1;
    sensorTermios.c_cc[VTIME] = 1;
    cfsetispeed (&sensorTermios, B9600);
    cfsetospeed (&sensorTermios, B9600);
    if (tcsetattr (sensorFd, TCSANOW, &sensorTermios) == -1)
        ErrSys ("tcsetattr");

    // Create the queues

    if (pipe (stateMachQue) == -1)
        ErrSys ("create State Machine Queue");
    stateMachQueWrFd = stateMachQue[1];

    // Initiate the Reply Receiver

    if ((replyRcvrPid = fork()) == -1)
        ErrSys ("initiate Reply Receiver");
    if (replyRcvrPid == 0)
        ReplyRcvr (sensorFd);

    // Initiate the Clock Tick Generator

    if ((clkTickGenPid = fork()) == -1)
        ErrSys ("initiate Clock Tick Generator");
    if (clkTickGenPid == 0)
        ClkTickGen ();

    // Initiate the State Machine

    StateMach (sensorFd, stateMachQue[0]);

    // Keep the compiler happy.

    return 0;
}
```