A Listings

Include EVERY source file of your Application (including headers)!!! And EVERY file provided by us which you have modified!

A.1 Application

Listing 1: ../RadioScannerAppC.nc

```
/**
 *
 * @file RadioScannerAppC.nc
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-12-09
 *
 * Top-level wiring of the RadioScanner app.
 *
**/

#include <debug.h>
configuration RadioScannerAppC {

}

implementation {
    components MainC, RadioScannerP, DatabaseC, FMClickC, PS2C, GlcdC, VolumeAdcC;
        components BufferedLcdC as Lcd;
        components new TimerMilliC() as VolumeTimer;
        components new TimerMilliC() as ErrorTimer;
        components new TimerMilliC() as RDSTimer;

        RadioScannerP.Boot -> MainC.Boot;
        RadioScannerP.Glcd -> GlcdC.Glcd;
        RadioScannerP.Lcd -> Lcd;
        RadioScannerP.DBInit -> DatabaseC.Init;
        RadioScannerP.DB -> DatabaseC.Database;
        RadioScannerP.RadioInit -> FMClickC.Init;
        RadioScannerP.Radio -> FMClickC.FMClick;
        RadioScannerP.Keyboard -> PS2C.PS2;
        RadioScannerP.volumeKnob -> VolumeAdcC.Read;
        RadioScannerP.VolumeTimer -> VolumeTimer;
        RadioScannerP.ErrorTimer -> ErrorTimer;
        RadioScannerP.RDSTimer -> RDSTimer;
}
```

Listing 2: ../RadioScannerP.nc

```
/*
 *
 * @file RadioScannerP.nc
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-12-20
 *
 * Implementation of the RadioScanner app.
 *
**/

#include <avr/pgmspace.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <text.h>

module RadioScannerP {
    uses
    {
```

```
        interface Boot;
        interface Glcd;
        interface BufferedLcd as Lcd;
        interface Init as DBInit;
        interface Database as DB;
        interface Init as RadioInit;
        interface FMClick as Radio;
        interface PS2 as Keyboard;
        interface Read<uint16_t> as volumeKnob;
        interface Timer<TMilli> as VolumeTimer;
        interface Timer<TMilli> as ErrorTimer;
        interface Timer<TMilli> as RDSTimer;
    }
}

implementation {
    #define BAND_LIMIT_LO    875
    #define BAND_LIMIT_HI    1080

    #define DISPLAY_UPDATE_RATE 1000
    #define VOLUME_UPDATE_RATE  100
    #define ERROR_MSG_TIMEOUT   700
    #define RDS_TIMEOUT         5000

    #define GLCD_CHARS_PER_LINE     21
    #define GLCD_CHAR_WIDTH         6
    #define GLCD_LEFT_END           0
    #define GLCD_FIRST_LINE         10
    #define GLCD_LINE_SPACE         10
    #define GLCD_TRUE_LEFT_END      122
    #define GLCD_TRUE_FIRST_LINE    7
    #define GLCD_TRUE_LINE_SPACE    8

    enum app_state {INIT, KBCTRL, TUNEINP, TUNE, SEEK, BANDSEEK, ADD, FAV, NOTE};
    static enum app_state appState;

    static char kbChar;
    static uint16_t currChan;
    static uint16_t nextChan;

    /* Buffer for frequency input */
    #define TUNEINPUT_BUF_SZ 5
    static struct
    {
        uint8_t idx;
        char buf[TUNEINPUT_BUF_SZ];
    } tuneInput;

    /* RDS data storage */
    #define PS_BUF_SZ   8
    #define RT_BUF_SZ   64
    #define CT_BUF_SZ   6
    static struct
    {
        bool PSAvail;
        bool newPS;
        bool newRT;
        bool newCT;
        uint16_t piCode;
        /* Need space for null termination */
        char PS[PS_BUF_SZ+1];
        char RT[RT_BUF_SZ+1];
        char CT[CT_BUF_SZ];
    } rds;

    /* Buffers for name and note are needed, as channelInfo struct only contains pointers */
    #define NAME_SZ 8
    #define NOTE_SZ 40
    typedef struct
    {
```

```
    channelInfo info;
    char name[NAME_SZ+1];
    char note[NOTE_SZ+1];
} channel_t;

static struct
{
    uint8_t idx;
    char buf[NOTE_SZ+1];
} noteInput;

/* The internal channel list */
#define CHANNEL_LIST_SZ 15
static struct
{
    uint8_t entries;
    channel_t list[CHANNEL_LIST_SZ];
} channels;

/* Table of favourite channels */
#define FAV_CNT         9
static struct
{
    uint8_t entries;
    uint8_t table[FAV_CNT];
} favourites;

static uint8_t oldVolume;
static uint8_t newVolume;
static uint8_t errno;

task void inputTuneChannel(void);
task void inputNote(void);
task void displayChannelInfo(void);
task void displayRDS(void);
task void setVolume(void);
task void startSeekUp(void);
task void startSeekDown(void);
task void startSeekBand(void);
task void startTune(void);
task void addChannel(void);
task void displayHardError(void);
task void displaySoftError(void);

static void printVolume(void);
static void clearRDSData(void);
static void addFavourite(void);
static uint8_t getListId(uint16_t channel);
static uint8_t getNextId(uint16_t channel);
static void tuneNextHighest(void);
static void addNote(void);
static void tuneToFavourite(char c);
static void removeIllegalChars(char *data, uint8_t len);

//////////////////////
/* Tasks           */
//////////////////////

task void handleChar(void)
{
    char c;
    atomic { c = kbChar; }

    switch (c)
    {
        /* Add/update current channel info */
        case 'a':
            atomic { appState = ADD; }
            post addChannel();
            break;
```

```
    /* Add current channel to favourites */
    case 'f':
        atomic { appState = FAV; }
        addFavourite();
        break;

    /* Tune to list entry with next highest frequency */
    case 'l':
        tuneNextHighest();
        break;

    /* Add note */
    case 'n':
        atomic { appState = NOTE; }
        addNote();
        break;

    case 's':
        call Radio.receiveRDS(FALSE);
        clearRDSData();
        call DB.purgeChannelList();
        favourites.entries = 0;
        memset(favourites.table, 0xff, FAV_CNT);
        atomic
        {
            channels.entries = 0;
            appState = BANDSEEK;
            nextChan = BAND_LIMIT_LO;
        }
        post startTune();
        break;

    /* Enter frequency and tune to channel */
    case 't':
        call Radio.receiveRDS(FALSE);
        clearRDSData();
        atomic
        {
            tuneInput.idx = 0;
            memset(tuneInput.buf, 0, TUNEINPUT_BUF_SZ);
            appState = TUNEINP;
        }
        call Glcd.fill(0x00);
        call Glcd.drawTextPgm(text_channelInput, GLCD_LEFT_END, GLCD_FIRST_LINE);
        post inputTuneChannel();
        break;

    /* Seek next higher channel */
    case '.':
        call Radio.receiveRDS(FALSE);
        clearRDSData();
        atomic { appState = SEEK; }
        post startSeekUp();
        break;

    /* Seek next lower channel */
    case ',':
        call Radio.receiveRDS(FALSE);
        clearRDSData();
        atomic { appState = SEEK; }
        post startSeekDown();
        break;

    default:
        /* Favourites access */
        if (isdigit(c))
            tuneToFavourite(c);
        break;
}
```

```
}

/*
 * @brief Read input from the keyboard and tune to the specified frequency.
 */
task void inputTuneChannel(void)
{
    char c;
    atomic { c = kbChar; }

    if (isdigit(c))
    {
        if (tuneInput.idx < TUNEINPUT_BUF_SZ-1)
            tuneInput.buf[tuneInput.idx++] = c;
    }

    /* Backspace */
    if (c == '\b' && tuneInput.idx > 0)
        tuneInput.buf[--tuneInput.idx] = '\0';

    call Glcd.drawTextPgm(text_emptyTime, GLCD_LEFT_END, GLCD_FIRST_LINE+GLCD_LINE_SPACE);
    call Glcd.drawText(tuneInput.buf, GLCD_LEFT_END, GLCD_FIRST_LINE+GLCD_LINE_SPACE);

    /* Complete entering channel */
    if (c == '\n')
    {
        uint16_t channel = (uint16_t) strtoul(tuneInput.buf, NULL, 10);
        atomic { appState = TUNE; }
        if (channel < BAND_LIMIT_LO || channel > BAND_LIMIT_HI)
        {
            atomic { errno = E_CHAN_INVAL; }
            post displaySoftError();
        }
        else
        {
            atomic { nextChan = channel; }
            post startTune();
        }
    }
}

/*
 * @brief Read input from the keyboard and save as note.
 */
task void inputNote(void)
{
    char c;
    char line[GLCD_CHARS_PER_LINE+1];
    atomic { c = kbChar; }

    /* Backspace */
    if (c == '\b')
    {
        if (noteInput.idx > 0)
            noteInput.buf[--noteInput.idx] = '\0';
    }
    else if (c != '\n' && c != '\r' && c != '\0')
    {
        if (noteInput.idx < NOTE_SZ)
            noteInput.buf[noteInput.idx++] = c;
    }

    /* Write input to screen */
    memcpy(line, noteInput.buf, GLCD_CHARS_PER_LINE);
    line[GLCD_CHARS_PER_LINE] = '\0';
    call Glcd.drawTextPgm(text_emptyLine, GLCD_TRUE_LEFT_END, GLCD_FIRST_LINE+
        ↪ GLCD_LINE_SPACE);
    call Glcd.drawText(line, GLCD_TRUE_LEFT_END, GLCD_FIRST_LINE+GLCD_LINE_SPACE);
    memcpy(line, noteInput.buf+GLCD_CHARS_PER_LINE, GLCD_CHARS_PER_LINE-2);
    line[GLCD_CHARS_PER_LINE-2] = '\0';
```

```
        call Glcd.drawTextPgm(text_emptyLine, GLCD_TRUE_LEFT_END, GLCD_FIRST_LINE+
            ↪ GLCD_LINE_SPACE*2);
        call Glcd.drawText(line, GLCD_TRUE_LEFT_END, GLCD_FIRST_LINE+GLCD_LINE_SPACE*2);

        /* Complete entering note */
        if (c == '\n')
        {
            uint8_t id;
            uint16_t channel;
            channel_t *ce;

            atomic
            {
                appState = ADD;
                channel = currChan;
            }

            id = getListId(channel);
            atomic { ce = &channels.list[id]; }

            noteInput.buf[noteInput.idx] = '\0';
            memset(ce->info.notes, 0, NOTE_SZ+1);
            strncpy(ce->info.notes, noteInput.buf, NOTE_SZ);
            call DB.saveChannel(id, &ce->info);
        }
    }

    /*
     * @brief Main task: display the radio station information.
     */
    task void displayChannelInfo(void)
    {
        char freqBuf[5], idBuf[4], line[GLCD_CHARS_PER_LINE+1];
        uint8_t id, qdial;
        uint16_t chan;
        enum app_state state;

        atomic
        {
            state = appState;
            chan = currChan;
        }

        id = getListId(chan);

        /* Display header */
        sprintf(freqBuf, "%d.%d", chan/10, chan%10);
        call Glcd.fill(0x00);
        call Glcd.drawText(freqBuf, GLCD_LEFT_END, GLCD_TRUE_FIRST_LINE);
        call Glcd.drawTextPgm(text_MHz, GLCD_CHAR_WIDTH*6, GLCD_TRUE_FIRST_LINE);

        /* Display list id and quick dial if channel is in list */
        if (id < CHANNEL_LIST_SZ)
        {
            sprintf(idBuf, "c%02d", id);
            idBuf[3] = '\0';
            call Glcd.drawText(idBuf, GLCD_CHAR_WIDTH*10, GLCD_TRUE_FIRST_LINE);

            qdial = channels.list[id].info.quickDial;
            if (qdial > 0)
            {
                sprintf(idBuf, "f%d", qdial);
                idBuf[2] = '\0';
                call Glcd.drawText(idBuf, GLCD_CHAR_WIDTH*14, GLCD_TRUE_FIRST_LINE);
            }

            /* Display note */
            memcpy(line, channels.list[id].note, GLCD_CHARS_PER_LINE);
            line[GLCD_CHARS_PER_LINE] = '\0';
            call Glcd.drawTextPgm(text_emptyLine, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
```

```
            ↪ GLCD_TRUE_LINE_SPACE*5);
        call Glcd.drawText(line, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE*5);
        memcpy(line, channels.list[id].note+GLCD_CHARS_PER_LINE, GLCD_CHARS_PER_LINE-2);
        line[GLCD_CHARS_PER_LINE-2] = '\0';
        call Glcd.drawTextPgm(text_emptyLine, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE*6);
        call Glcd.drawText(line, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE*6);
    }

    /* Display initial RDS data */
    call Glcd.drawTextPgm(text_emptyName, GLCD_LEFT_END, GLCD_TRUE_FIRST_LINE+
        ↪ GLCD_TRUE_LINE_SPACE);
    call Glcd.drawText(rds.PS, GLCD_LEFT_END, GLCD_TRUE_FIRST_LINE+GLCD_TRUE_LINE_SPACE);
    call Glcd.drawTextPgm(text_emptyTime, GLCD_LEFT_END, GLCD_TRUE_FIRST_LINE+
        ↪ GLCD_TRUE_LINE_SPACE);
    call Glcd.drawText(rds.CT, GLCD_CHAR_WIDTH*9, GLCD_TRUE_FIRST_LINE+
        ↪ GLCD_TRUE_LINE_SPACE);
    memcpy(line, rds.RT, GLCD_CHARS_PER_LINE);
    line[GLCD_CHARS_PER_LINE] = '\0';
    call Glcd.drawTextPgm(text_emptyLine, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
        ↪ GLCD_TRUE_LINE_SPACE*2);
    call Glcd.drawText(line, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+GLCD_TRUE_LINE_SPACE
        ↪ *2);
    memcpy(line, rds.RT+GLCD_CHARS_PER_LINE, GLCD_CHARS_PER_LINE);
    line[GLCD_CHARS_PER_LINE] = '\0';
    call Glcd.drawTextPgm(text_emptyLine, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
        ↪ GLCD_TRUE_LINE_SPACE*3);
    call Glcd.drawText(line, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+GLCD_TRUE_LINE_SPACE
        ↪ *3);
    memcpy(line, rds.RT+GLCD_CHARS_PER_LINE*2, GLCD_CHARS_PER_LINE);
    line[GLCD_CHARS_PER_LINE] = '\0';
    call Glcd.drawTextPgm(text_emptyLine, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
        ↪ GLCD_TRUE_LINE_SPACE*4);
    call Glcd.drawText(line, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+GLCD_TRUE_LINE_SPACE
        ↪ *4);

    call Radio.receiveRDS(TRUE);

    if (BANDSEEK == state)
        call RDSTimer.startOneShot(RDS_TIMEOUT);
}

/*
 * @brief Update the RDS information on screen.
 */
task void displayRDS(void)
{
    bool newPS, newRT, newCT;

    atomic
    {
        newPS = rds.newPS;
        newRT = rds.newRT;
        newCT = rds.newCT;
    }

    if (newPS)
    {
        removeIllegalChars(rds.PS, PS_BUF_SZ);
        call Glcd.drawTextPgm(text_emptyName, GLCD_LEFT_END, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE);
        call Glcd.drawText(rds.PS, GLCD_LEFT_END, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE);
        atomic { rds.newPS = FALSE; }
    }
    if (newCT)
    {
        call Glcd.drawTextPgm(text_emptyTime, GLCD_LEFT_END, GLCD_TRUE_FIRST_LINE+
```

```
            ↪ GLCD_TRUE_LINE_SPACE);
        call Glcd.drawText(rds.CT, GLCD_CHAR_WIDTH*9, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE);
        atomic { rds.newCT = FALSE; }
    }
    if (newRT)
    {
        char line[GLCD_CHARS_PER_LINE+1];

        removeIllegalChars(rds.RT, RT_BUF_SZ);
        memcpy(line, rds.RT, GLCD_CHARS_PER_LINE);
        line[GLCD_CHARS_PER_LINE] = '\0';
        call Glcd.drawTextPgm(text_emptyLine, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE*2);
        call Glcd.drawText(line, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE*2);
        memcpy(line, rds.RT+GLCD_CHARS_PER_LINE, GLCD_CHARS_PER_LINE);
        line[GLCD_CHARS_PER_LINE] = '\0';
        call Glcd.drawTextPgm(text_emptyLine, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE*3);
        call Glcd.drawText(line, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE*3);
        memcpy(line, rds.RT+GLCD_CHARS_PER_LINE*2, GLCD_CHARS_PER_LINE);
        line[GLCD_CHARS_PER_LINE] = '\0';
        call Glcd.drawTextPgm(text_emptyLine, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE*4);
        call Glcd.drawText(line, GLCD_TRUE_LEFT_END, GLCD_TRUE_FIRST_LINE+
            ↪ GLCD_TRUE_LINE_SPACE*4);

        atomic { rds.newRT = FALSE; }
    }
}

/*
 * @brief Set the volume on the FMClick board if the value has changed.
 */
task void setVolume(void)
{
    if (newVolume != oldVolume)
    {
        call Radio.setVolume(newVolume);
        oldVolume = newVolume;
        printVolume();
    }
}

/*
 * @brief Start seeking upwards.
 */
task void startSeekUp(void)
{
    if (call Radio.seek(UP) != SUCCESS)
        post startSeekUp();
}

/*
 * @brief Start seeking downwards.
 */
task void startSeekDown(void)
{
    if (call Radio.seek(DOWN) != SUCCESS)
        post startSeekDown();
}

/*
 * @brief Start seeking the whole band.
 */
task void startSeekBand(void)
{
    if (call Radio.seek(BAND) != SUCCESS)
```

```
        post startSeekBand();
}

/*
 * @brief Start tuning to a specific frequency.
 */
task void startTune(void)
{
    uint16_t channel;
    atomic { channel = nextChan; }
    if (call Radio.tune(channel) != SUCCESS)
        post startTune();
}

/*
 * @brief Add/update the current channel to the local list and the DB.
 */
task void addChannel(void)
{
    uint8_t id;
    uint16_t freq;
    atomic { freq = currChan; }

    call Radio.receiveRDS(FALSE);
    id = getListId(freq);

    /* Channel already in list, update */
    if (id < CHANNEL_LIST_SZ)
    {
        bool PSAvail;
        channel_t *c;

        atomic
        {
            PSAvail = rds.PSAvail;
            c = &channels.list[channels.entries++];
        }

        if (PSAvail)
        {
            memset(c->info.name, 0, NAME_SZ+1);
            removeIllegalChars(rds.PS, PS_BUF_SZ);
            snprintf(c->info.name, NAME_SZ, "%-8s", rds.PS);
            atomic { c->info.pi_code = rds.piCode; }
            call DB.saveChannel(id, &c->info);
        }
        /* Nothing to update */
        else
        {
            atomic {appState = KBCTRL; }
            post displayChannelInfo();
        }
    }
    /* Add channel to list */
    else
    {
        if (channels.entries >= CHANNEL_LIST_SZ)
        {
            atomic { errno = E_LIST_FULL; }
            post displaySoftError();
        }
        else
        {
            bool PSAvail;
            channel_t *c;

            atomic
            {
                PSAvail = rds.PSAvail;
                c = &channels.list[channels.entries++];
```

```
            }

            c->info.quickDial = 0;
            c->info.frequency = freq;
            c->info.pi_code = 1;
            c->info.name = c->name;
            c->info.notes = c->note;
            memset(c->info.name, 0, NAME_SZ+1);
            memset(c->info.notes, 0, NOTE_SZ+1);

            if (PSAvail)
            {
                atomic { c->info.pi_code = rds.piCode; }
                snprintf(c->info.name, NAME_SZ, "%-8s", rds.PS);
            }
            /* Add note if no RDS data available */
            else
                strcpy_P(c->info.notes, text_noRDS);

            call DB.saveChannel(0xff, &c->info);
        }
    }
}

/*
 * @brief Display an error message according to errno and remain in this state forever.
 */
task void displayHardError(void)
{
    uint8_t err;
    atomic { err = errno; }

    call Radio.receiveRDS(FALSE);
    call Glcd.fill(0x00);
    call Glcd.drawTextPgm(text_error, GLCD_LEFT_END, GLCD_FIRST_LINE);

    switch (err)
    {
        case E_FMCLICK_INIT:
            call Glcd.drawTextPgm(text_FMClickInitFail, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;

        default:
            call Glcd.drawTextPgm(text_unknownError, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;
    }
}

/*
 * @brief Display an error message according to errno and return to previous state.
 */
task void displaySoftError(void)
{
    uint8_t err;
    atomic { err = errno; }

    call Radio.receiveRDS(FALSE);
    call Glcd.fill(0x00);
    call Glcd.drawTextPgm(text_error, GLCD_LEFT_END, GLCD_FIRST_LINE);

    switch (err)
    {
        case E_CHAN_INVAL:
            call Glcd.drawTextPgm(text_chanInval, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;

        case E_LIST_FULL:
```

```
            call Glcd.drawTextPgm(text_listFull, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;

        case E_FAVS_FULL:
            call Glcd.drawTextPgm(text_favsFull, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;

        case E_DB_FULL:
            call Glcd.drawTextPgm(text_dbFull, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;

        case E_DB_ERR:
            call Glcd.drawTextPgm(text_dbErr, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;

        case E_BAND_LIMIT:
            call Glcd.drawTextPgm(text_bandLimit, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;

        case E_FAV_NSET:
            call Glcd.drawTextPgm(text_favNset, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;

        case E_CHAN_NLIST:
            call Glcd.drawTextPgm(text_chanNlist, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;

        case E_IS_FAV:
            call Glcd.drawTextPgm(text_isFav, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;

        default:
            call Glcd.drawTextPgm(text_unknownError, GLCD_LEFT_END, GLCD_FIRST_LINE+
                ↪ GLCD_LINE_SPACE);
            break;
    }

    call ErrorTimer.startOneShot(ERROR_MSG_TIMEOUT);
}

//////////////////////
/* Internal functions */
//////////////////////

/*
 * @brief Print the current volume value on the lcd screen.
 */
static void printVolume(void)
{
    char volBuf[3];

    sprintf(volBuf, "%02d", oldVolume);
    volBuf[2] = '\0';

    call Lcd.goTo(0,8);
    call Lcd.write(volBuf);
    call Lcd.forceRefresh();
}

/*
 * @brief Clear the RDS buffers.
 */
```

```
static void clearRDSData(void)
{
    atomic
    {
        rds.PSAvail = FALSE;
        rds.newPS = FALSE;
        rds.newRT = FALSE;
        rds.newCT = FALSE;
    }
    memset(rds.PS, 0, PS_BUF_SZ+1);
    memset(rds.RT, 0, RT_BUF_SZ+1);
    memset(rds.CT, 0, CT_BUF_SZ);
}

/*
 * @brief Add the current channel to the favourites list.
 */
static void addFavourite(void)
{
    if (favourites.entries < FAV_CNT)
    {
        uint8_t id;
        uint16_t chan;

        atomic { chan = currChan; }
        id = getListId(chan);

        if (id < CHANNEL_LIST_SZ)
        {
            /* Channel already in favourites */
            if (channels.list[id].info.quickDial > 0)
            {
                atomic { errno = E_IS_FAV; }
                post displaySoftError();
            }
            else
            {
                uint8_t fid;

                /* Search free quick dial slot */
                for (fid = 0; fid < FAV_CNT; fid++)
                {
                    if (favourites.table[fid] == 0xff)
                        break;
                }

                favourites.table[fid] = id;
                favourites.entries++;
                atomic { channels.list[id].info.quickDial = fid+1; }
                call DB.saveChannel(id, &channels.list[id].info);
            }
        }
        /* Channel not in list */
        else
        {
            atomic { errno = E_CHAN_NLIST; }
            post displaySoftError();
        }
    }
    /* Favourites full */
    else
    {
        atomic { errno = E_FAVS_FULL; }
        post displaySoftError();
    }
}

/*
 * @brief           Check if the specified channel has been saved to the list.
 * @param channel   Channel frequency to check.
```

```
 * @return          Return the channel ID if successful or 0xff on failure.
 */
static uint8_t getListId(uint16_t channel)
{
    uint8_t id;

    for (id = 0; id < channels.entries; id++)
    {
        if (channels.list[id].info.frequency == channel)
            return id;
    }

    return 0xff;
}

/*
 * @brief           Get the ID of the channel with the next highest frequency in the list.
 * @param channel   Channel frequency to start from.
 * @return          Return the channel ID if successful or 0xff on failure.
 */
static uint8_t getNextId(uint16_t channel)
{
    uint8_t id, next;
    uint16_t currDist, minDist;

    next = 0xff;
    minDist = 0xffff;
    for (id = 0; id < channels.entries; id++)
    {
        currDist = channels.list[id].info.frequency - channel;
        if (currDist > 0 && currDist < minDist)
        {
            next = id;
            minDist = currDist;
        }
    }

    return next;
}

/*
 * @brief Tune to the channel with the next highest frequency.
 */
static void tuneNextHighest(void)
{
    if (channels.entries > 0)
    {
        uint8_t nextId;
        uint16_t channel;

        atomic { channel = currChan; }
        nextId = getNextId(channel);

        if (nextId < CHANNEL_LIST_SZ)
        {
            call Radio.receiveRDS(FALSE);
            clearRDSData();
            atomic
            {
                appState = TUNE;
                nextChan = channels.list[nextId].info.frequency;
            }
            post startTune();
        }
    }
}

/*
 * @brief Start adding a note to a channel.
 */
```

```
static void addNote(void)
{
    uint8_t id;
    uint16_t channel;

    atomic { channel = currChan; }
    id = getListId(channel);

    if (id < CHANNEL_LIST_SZ)
    {
        call Radio.receiveRDS(FALSE);
        atomic { kbChar = '\0'; }
        noteInput.idx = 0;
        memset(noteInput.buf, 0, NOTE_SZ+1);
        call Glcd.fill(0x00);
        call Glcd.drawTextPgm(text_noteInput, GLCD_LEFT_END, GLCD_FIRST_LINE);
        post inputNote();
    }
    else
    {
        atomic { errno = E_CHAN_NLIST; }
        post displaySoftError();
    }
}

/*
 * @brief Access the favourite stations.
 */
static void tuneToFavourite(char c)
{
    uint8_t fav = (uint8_t)(c - '0');
    if (fav > 0 && fav <= 9)
    {
        uint8_t favId = favourites.table[fav-1];

        if (favId < CHANNEL_LIST_SZ)
        {
            call Radio.receiveRDS(FALSE);
            clearRDSData();
            atomic
            {
                appState = TUNE;
                nextChan = channels.list[favId].info.frequency;
            }
            post startTune();
        }
        else
        {
            atomic { errno = E_FAV_NSET; }
            post displaySoftError();
        }
    }
}

/*
 * @brief Remove '\n' and '\r' from a string.
 */
static void removeIllegalChars(char *data, uint8_t len)
{
    uint8_t i;

    for (i = 0; i < len; i++)
    {
        if (data[i] == '\n' || data[i] == '\r')
            data[i] = ' ';
    }
}

//////////////////////
/* Events */
```

```
//////////////////////

/*
 * @brief Inititalize all modules.
 */
event void Boot.booted()
{
    char textBuf[8];
    uint8_t id;
    channel_t *c;

    /* Initialize list datastructure */
    for (id = 0; id < CHANNEL_LIST_SZ; id++)
    {
        atomic { c = &channels.list[id]; }
        c->info.name = c->name;
        c->info.notes = c->note;
        memset(c->info.name, 0, NAME_SZ+1);
        memset(c->info.notes, 0, NOTE_SZ+1);
    }

    atomic
    {
        appState = INIT;
        channels.entries = 0;
    }

    favourites.entries = 0;
    memset(favourites.table, 0xff, FAV_CNT);

    oldVolume = 0;
    newVolume = 0;

    strcpy_P(textBuf, text_volume);
    textBuf[7] = '\0';
    call Lcd.goTo(0,0);
    call Lcd.write(textBuf);
    printVolume();

    call Glcd.fill(0x00);
    call Glcd.drawTextPgm(text_init, GLCD_LEFT_END, GLCD_FIRST_LINE);

    call Keyboard.init();
    call RadioInit.init();
    call DBInit.init();
}

/*
 * @brief If FMClick is ready, we are good to go.
 */
event void Radio.initDone(error_t res)
{
    if (res == SUCCESS)
    {
        /* Set volume and fetch database entries */
        call Radio.receiveRDS(FALSE);
        call volumeKnob.read();
        call VolumeTimer.startPeriodic(VOLUME_UPDATE_RATE);
        call DB.getChannelList(FALSE);
    }
    else
    {
        atomic { errno = E_FMCLICK_INIT; }
        post displayHardError();
    }
}

/*
 * @brief Handle keyboard input.
 */
```

```
async event void Keyboard.receivedChar(uint8_t c)
{
    enum app_state state;

    atomic
    {
        state = appState;
        kbChar = c;
    }

    switch (state)
    {
        case KBCTRL:
            post handleChar();
            break;

        case TUNEINP:
            post inputTuneChannel();
            break;

        case NOTE:
            post inputNote();
            break;

        default:
            break;
    }
}

/*
 * @brief Select action after radio tuned to station.
 */
async event void Radio.tuneComplete(uint16_t channel)
{
    enum app_state state;

    atomic
    {
        currChan = channel;
        state = appState;
    }

    switch (state)
    {
        case BANDSEEK:
            post startSeekBand();
            break;

        case TUNE:
            atomic { appState = KBCTRL; }
            post displayChannelInfo();
            break;

        default:
            atomic { appState = KBCTRL; }
            post displayChannelInfo();
            break;
    }
}

/*
 * @brief Select action after radio seek complete.
 */
async event void Radio.seekComplete(uint16_t channel)
{
    enum app_state state;
    atomic
    {
        currChan = channel;
        state = appState;
```

```
    }

    switch (state)
    {
        case BANDSEEK:
            if (channel <= BAND_LIMIT_HI)
                post displayChannelInfo();
            else
            {
                if (channels.entries > 0)
                {
                    clearRDSData();
                    atomic
                    {
                        nextChan = channels.list[0].info.frequency;
                        appState = TUNE;
                    }
                    post startTune();
                }
                else
                {
                    atomic { appState = KBCTRL; }
                    post displayChannelInfo();
                }
            }
            break;

        case SEEK:
            atomic { appState = KBCTRL; }
            post displayChannelInfo();
            break;

        default:
            atomic { appState = KBCTRL; }
            post displayChannelInfo();
            break;
    }
}

/*
 * @brief Grab the RDS data from the driver.
 */
async event void Radio.rdsReceived(RDSType type, char *buf)
{
    enum app_state state;
    atomic { state = appState; }

    switch (type)
    {
        case PS:
            memset(rds.PS, 0, PS_BUF_SZ+1);
            memcpy(rds.PS, buf, PS_BUF_SZ);
            atomic
            {
                rds.PSAvail = TRUE;
                rds.newPS = TRUE;
                rds.piCode = ((uint16_t)buf[PS_BUF_SZ+1]) & 0x00ff;
                rds.piCode |= ((uint16_t)buf[PS_BUF_SZ]) << 8;
            }
            if (BANDSEEK == state)
                post addChannel();
            break;

        case RT:
            atomic { rds.newRT = TRUE; }
            memset(rds.RT, 0, RT_BUF_SZ+1);
            memcpy(rds.RT, buf, RT_BUF_SZ);
            break;

        case TIME:
```

```
            atomic { rds.newCT = TRUE; }
            memset(rds.CT, 0, CT_BUF_SZ);
            memcpy(rds.CT, buf, CT_BUF_SZ);
            break;

        default:
            break;
    }

    post displayRDS();
}

event void VolumeTimer.fired()
{
    call volumeKnob.read();
}

/*
 * @brief Timeout of soft error message.
 */
event void ErrorTimer.fired()
{
    atomic { appState = KBCTRL; }
    post displayChannelInfo();
}

/*
 * @brief RDS timeout for band seek.
 */
event void RDSTimer.fired()
{
    post addChannel();
}

/*
 * @brief ADC conversion finished.
 */
event void volumeKnob.readDone(error_t res, uint16_t val)
{
    newVolume = (uint8_t)(val >> 6);
    post setVolume();
}

/*
 * @brief Process channel entry from the DB.
 */
event void DB.receivedChannelEntry(uint8_t id, channelInfo channel)
{
    enum app_state state;
    atomic { state = appState; }

    if (INIT == state)
    {
        /* Received last DB entry */
        if (id == 0xff)
        {
            /* Tune to first channel in list */
            if (channels.entries > 0)
            {
                clearRDSData();
                atomic
                {
                    nextChan = channels.list[0].info.frequency;
                    appState = TUNE;
                }
                post startTune();
            }
            /* DB empty - start band seek */
            else
            {
```

```
                clearRDSData();
                call DB.purgeChannelList();
                favourites.entries = 0;
                memset(favourites.table, 0xff, FAV_CNT);
                atomic
                {
                    channels.entries = 0;
                    appState = BANDSEEK;
                    nextChan = BAND_LIMIT_LO;
                }
                post startTune();
            }
        }
        /* Add entry to the list */
        else
        {
            channel_t *c;
            atomic { c = &channels.list[channels.entries++]; }

            memcpy(&c->info, &channel, sizeof(channelInfo));
            c->info.name = c->name;
            c->info.notes = c->note;
            snprintf(c->info.name, NAME_SZ, "%-8s", channel.name);
            strncpy(c->info.notes, channel.notes, NOTE_SZ);

            /* Add entry to favourites */
            if (c->info.quickDial > 0)
            {
                favourites.table[c->info.quickDial-1] = channels.entries-1;
                favourites.entries++;
            }
        }
    }
}

/*
 * @brief Check if channel has been saved suceessfully to the DB.
 */
event void DB.savedChannel(uint8_t id, uint8_t result)
{
    uint8_t state;
    atomic { state = appState; }

    if (result == 0)
    {
        switch (state)
        {
            case ADD:
                atomic { appState = KBCTRL; }
                post displayChannelInfo();
                break;

            case FAV:
                atomic { appState = KBCTRL; }
                post displayChannelInfo();
                break;

            case BANDSEEK:
                clearRDSData();
                post startSeekBand();
                break;

            default:
                atomic { appState = KBCTRL; }
                post displayChannelInfo();
                break;
        }
    }
    else if (result == 1)
    {
```

```
            atomic { errno = E_DB_FULL; }
            post displaySoftError();
        }
        else
        {
            atomic { errno = E_DB_ERR; }
            post displaySoftError();
        }
    }
}
```

Listing 3: ../text.h

```
/**
 *
 * @file text.h
 * @author  Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date    2019-01-13
 *
 * Display text for the main application, stored in PROGMEM.
 *
**/

#ifndef __TEXT__
#define __TEXT__

#include <avr/pgmspace.h>

#define CHARS_PER_LINE  21

/* Application messages */

const char text_channelInput[] PROGMEM = "Enter Channel:";
const char text_noteInput[] PROGMEM = "Enter Note:";
const char text_channel[] PROGMEM = "Channel: ";
const char text_MHz[] PROGMEM = "MHz";
const char text_volume[] PROGMEM = "Volume:";
const char text_emptyName[] PROGMEM = "        ";
const char text_emptyTime[] PROGMEM = "     ";
const char text_emptyLine[] PROGMEM = "                    ";
const char text_noRDS[] PROGMEM = "No RDS data!";
const char text_init[] PROGMEM = "Initializing...";

/* Error messages */
const char text_error[] PROGMEM = "Error:";
const char text_unknownError[] PROGMEM = "Unknown error!";
#define E_FMCLICK_INIT 0
const char text_FMClickInitFail[] PROGMEM = "FMClick init failed!";
#define E_CHAN_INVAL 1
const char text_chanInval[] PROGMEM = "Invalid channel!";
#define E_LIST_FULL 2
const char text_listFull[] PROGMEM = "Channel list full!";
#define E_FAVS_FULL 3
const char text_favsFull[] PROGMEM = "Favourites full!";
#define E_DB_FULL 4
const char text_dbFull[] PROGMEM = "DB full!";
#define E_DB_ERR 5
const char text_dbErr[] PROGMEM = "DB error!";
#define E_BAND_LIMIT 6
const char text_bandLimit[] PROGMEM = "Band limit reached!";
#define E_FAV_NSET 7
const char text_favNset[] PROGMEM = "Favourite not set!";
#define E_CHAN_NLIST 8
const char text_chanNlist[] PROGMEM = "Channel not in list!";
#define E_IS_FAV 9
const char text_isFav[] PROGMEM = "Already favourite!";

#endif
```

A.2 Database

Listing 4: ../Database/Database.nc

```
//#include <database.h>

typedef struct {
    // The quick dial key [1..9], pass 0 for no/deleting quick dial
    uint8_t quickDial;

    // The channels's frequency in multiplies of 100kHZ, for example, 99.9MHZ =
    // 999 * 100kHz => 999
    uint16_t frequency;

    // The RDS PI Program Idenfitifaction code
    uint16_t pi_code;

    // The channel name, length maximal 9 characters, normally the RDS PS field
    char *name;

    // optional notes of the channel (length maximal 40 characters) pass NUL
    // (\0) to delete exisiting notes or set none on new entries
    char *notes;
} channelInfo;


interface Database
{
    /**
     * Save a new channel, or change properties of an existing one.
     * @param id The channel index from the database store, 0xFF to autoselect,
     *           must be between 0 and 15 if passed manually
     * @param channel The channel information, see channelInfo typedef
     */
    command void saveChannel(uint8_t id, channelInfo *channel);


    /**
     * Request the channel list from the database server
     * Received channels will be signaled through receivedChannelEntry
     * @param onlyFavorites tells server to send only the channels with a
     *        registered quickDial number, if not zero
     */
    command void getChannelList(uint8_t onlyFavorites);

    /**
     * Request the channel list from the database server
     * Received channels will be signaled through receivedChannelEntry
     */
    command void getChannel(uint8_t id);

    /**
     * Request that the Database purges all channels and their state
     * Received channels will be signaled through receivedChannelEntry
     */
    command void purgeChannelList();

    /**
     * Received highscore entry from the server.
     * @param id The channel index from the database store
     * @param channel The channel information, see channelInfo typedef
     */
    event void receivedChannelEntry(uint8_t id, channelInfo channel);

    /**
     * Server proceesed our request to save a Channel
     * @param id The channel index from the database store, the one we passed
     *           or the which was choosen if 0xFF was passed.
     * @param result 0 = OK, 1 = No free index (only ID auto choose), 2 = DB error
     */
```

```
    event void savedChannel(uint8_t id, uint8_t result);
}
```

Listing 5: ../Database/DatabaseC.nc

```
/**
 *
 * @file     DatabaseC.nc
 * @author   Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date     2018-12-26
 *
 * Configuration of the Database module.
 *
**/

#include <udp_config.h>

configuration DatabaseC {
    provides
    {
        interface Init;
        interface Database;
    }
}

implementation {
    components DatabaseP, IpTransceiverC, LlcTransceiverP;
    components Enc28j60C as EthernetC;
    components new UdpC(UDP_PORT);

    Init = DatabaseP.Init;
    Database = DatabaseP.Database;

    LlcTransceiverP.Mac -> EthernetC;

    DatabaseP.UdpSend -> UdpC;
    DatabaseP.UdpReceive -> UdpC;
    DatabaseP.IpControl -> IpTransceiverC;
    DatabaseP.Control -> EthernetC;
}
```

Listing 6: ../Database/DatabaseP.nc

```
/**
 *
 * @file     DatabaseP.nc
 * @author   Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date     2018-12-26
 *
 * Database module implementation.
 *
**/

#include <avr/pgmspace.h>
#include <ip.h>
#include <stdio.h>
#include <string.h>
#include <udp_config.h>
#include <commands.h>

module DatabaseP {
    provides
    {
        interface Init;
        interface Database;
    }
    uses
    {
        interface UdpSend;
```

```
        interface UdpReceive;
        interface IpControl;
        interface SplitControl as Control;
    }
}

implementation {
    #define NAME_MAX_LEN     8
    #define NOTE_MAX_LEN     40
    #define ERR_MSG_MAX_LEN  20

    #define DB_MAX_ENTRIES   15

    enum db_state {IDLE, INIT, ADD, LIST, GET, PURGE};
    static enum db_state dbState;
    static bool getList;
    static bool favOnly;

    /* Buffer for UDP messages */
    static struct
    {
        udp_msg_t send;
        udp_msg_t recv;
    } msgBuf;

    static struct
    {
        uint8_t entries;
        uint8_t currId;
        uint8_t ids[DB_MAX_ENTRIES];
    } list;

    typedef struct
    {
        char add[CMD_ADD_LEN+1];
        char update[CMD_UPDATE_LEN+1];
        char id[CMD_ID_LEN+1];
        char freq[CMD_FREQ_LEN+1];
        char qdial[CMD_QDIAL_LEN+1];
        char picode[CMD_PICODE_LEN+1];
        char name[CMD_NAME_LEN+1];
        char note[CMD_NOTE_LEN+1];
    } params_t;

    /* Task prototypes */
    task void sendTask(void);
    task void recvTask(void);
    task void fetchList(void);

    /* Function prototypes */
    static void decodeMessage(udp_msg_t *msg);
    static void decodeAdd(udp_msg_t *msg);
    static void decodeList(udp_msg_t *msg);
    static void decodeGet(udp_msg_t *msg);
    static bool prepareMessage(udp_msg_t *msg, uint8_t **paramStart);
    static bool parseChannelInfo(char *params, channelInfo *channel, uint8_t *id);
    static void getParamsPgm(params_t *p);

    //////////////////////
    /* Interface commands */
    //////////////////////

    /*
     * @brief Initialize the ethernet board.
     */
    command error_t Init.init(void)
    {
        in_addr_t *ip;
        in_addr_t cip = { .bytes {IP}};
        in_addr_t cnm = { .bytes {NETMASK}};
```

```
    in_addr_t cgw = { .bytes {GATEWAY}};

    atomic { dbState = INIT; }

    call IpControl.setIp(&cip);
    call IpControl.setNetmask(&cnm);
    call IpControl.setGateway(&cgw);

    ip = call IpControl.getIp();

    getList = FALSE;

    call Control.start();

    atomic { dbState = IDLE; }

    return SUCCESS;
}

/**
 * @brief           Save/update a channel in the DB.
 * @param id        The channel index from the database to update, 0xFF to autoselect for
    ↪ storing.
 * @param channel   The channel information, see channelInfo typedef.
 */
command void Database.saveChannel(uint8_t id, channelInfo *channel)
{
    enum db_state state;
    params_t params;
    atomic { state = dbState; }

    if (IDLE != state)
        return;

    atomic { dbState = ADD; }

    getParamsPgm(&params);
    memset(msgBuf.send.data, 0, MAX_MSG_LEN);

    /* Compose udp message */
    if (id == 0xff)
    {
        sprintf((char *) msgBuf.send.data, "%s\r%s=%d,%s=%-8s,%s=%d,%s=%d,%s=%d,%s=%s\n",
                params.add, params.id, 0, params.name, channel->name, params.qdial,
                channel->quickDial, params.freq, channel->frequency, params.picode,
                channel->pi_code, params.note, channel->notes);
    }
    else if (id < DB_MAX_ENTRIES)
    {
        sprintf((char *) msgBuf.send.data, "%s\r%s=%d,%s=%-8s,%s=%d,%s=%d,%s=%d,%s=%s\n",
                params.update, params.id, id, params.name, channel->name, params.qdial,
                channel->quickDial, params.freq, channel->frequency, params.picode,
                channel->pi_code, params.note, channel->notes);
    }

    msgBuf.send.data[MAX_MSG_LEN-1] = '\0';
    msgBuf.send.len = strlen((char *) msgBuf.send.data);

    post sendTask();
}

/**
 * @brief               Request the channel list from the database server, received
 *                      channels will be signaled through receivedChannelEntry.
 * @param onlyFavorites Tells server to send only the channels with a registered
 *                      quickDial number, if not zero.
 */
command void Database.getChannelList(uint8_t onlyFavorites)
{
    enum db_state state;
```

```
    char listp[CMD_LIST_LEN+1];
    atomic { state = dbState; }

    if (IDLE != state)
        return;

    atomic { dbState = LIST; }

    favOnly = onlyFavorites;
    strncpy_P(listp, cmd_list, CMD_LIST_LEN+1);
    memset(msgBuf.send.data, 0, MAX_MSG_LEN);

    /* Compose udp message */
    sprintf((char *) msgBuf.send.data, "%s\r\n", listp);
    msgBuf.send.len = strlen((char *) msgBuf.send.data);

    post sendTask();
}

/**
 * @brief       Request the channel list from the database server, received channels
 *              will be signaled through receivedChannelEntry.
 * @param id    The channel to load.
 */
command void Database.getChannel(uint8_t id)
{
    enum db_state state;
    char get[CMD_GET_LEN+1], idp[CMD_ID_LEN+1];
    atomic { state = dbState; }

    if (IDLE != state)
        return;

    atomic { dbState = GET; }

    strncpy_P(get, cmd_get, CMD_GET_LEN+1);
    strncpy_P(idp, cmd_id, CMD_ID_LEN+1);
    memset(msgBuf.send.data, 0, MAX_MSG_LEN);

    /* Compose udp message */
    sprintf((char *) msgBuf.send.data, "%s\r%s=%d\n", get, idp, id);
    msgBuf.send.len = strlen((char *) msgBuf.send.data);

    post sendTask();
}

/**
 * @brief   Request that the Database purges all channels and their state.
 */
command void Database.purgeChannelList()
{
    enum db_state state;
    char purge[CMD_PURGEALL_LEN+1];
    atomic { state = dbState; }

    if (IDLE != state)
        return;

    atomic { dbState = PURGE; }

    strncpy_P(purge, cmd_purgeall, CMD_PURGEALL_LEN+1);
    memset(msgBuf.send.data, 0, MAX_MSG_LEN);

    /* Compose udp message */
    sprintf((char *) msgBuf.send.data, "%s\r\n", purge);
    msgBuf.send.len = strlen((char *) msgBuf.send.data);

    post sendTask();
}
```

```
//////////////////////
/* Tasks              */
//////////////////////

/*
 * @brief Send a UDP message.
 */
task void sendTask(void)
{
    in_addr_t destination = { .bytes {DESTINATION}};

    if (call UdpSend.send(&destination, UDP_PORT, msgBuf.send.data, msgBuf.send.len) !=
        ↪ SUCCESS)
        post sendTask();
}

/*
 * @brief Process an incoming message.
 */
task void recvTask(void)
{
    decodeMessage(&msgBuf.recv);
    atomic { dbState = IDLE; }
}

/*
 * @brief Fetch the channel list from the DB.
 */
task void fetchList(void)
{
    if (list.entries - list.currId > 0)
        call Database.getChannel(list.ids[list.currId++]);
    else /* Already received last entry */
    {
        channelInfo dummy;
        getList = FALSE;
        list.entries = 0;
        list.currId = 0;
        atomic { dbState = IDLE; }
        signal Database.receivedChannelEntry(0xff, dummy);
    }
}

//////////////////////
/* Internal functions */
//////////////////////

/*
 * @brief       Decide how to decode an incoming message.
 * @param msg   The message to decode.
 */
static void decodeMessage(udp_msg_t *msg)
{
    enum db_state state;
    atomic { state = dbState; }

    switch (state)
    {
        case ADD:
            decodeAdd(msg);
            break;

        case LIST:
            decodeList(msg);
            break;

        case GET:
            decodeGet(msg);
            if (getList)
                post fetchList();
```

```
            break;

        case PURGE:
            break;

        default:
            return;
    }
}

/*
 * @brief       Decode add command response and signal result.
 * @param msg   The message to decode.
 */
static void decodeAdd(udp_msg_t *msg)
{
    uint8_t *paramStart;

    /* Invalid message */
    if (!prepareMessage(msg, &paramStart))
        return;

    /* OK message */
    if (strncmp_P((char *) msg->data, cmd_ok, CMD_OK_LEN) == 0)
    {
        char *k, *v;
        uint8_t id;

        k = strtok_r((char *) paramStart, "=", &v);
        id = (uint8_t) strtoul(v, NULL, 10);

        if (id < DB_MAX_ENTRIES)
            signal Database.savedChannel(id, 0);
        else
            signal Database.savedChannel(0, 2);
    }
    /* Error */
    else
    {
        uint8_t res = 2;
        char *_msg = strstr_P((char *) paramStart, cmd_msg);

        if (_msg != NULL)
        {
            /* Move to actual message */
            _msg += 4;

            if (strncmp_P(_msg, cmd_dbFull, CMD_DBFULL_LEN) == 0)
                res = 1;
        }

        signal Database.savedChannel(0, res);
    }
}

/*
 * @brief       Decode list command response.
 * @param msg   The message to decode.
 */
static void decodeList(udp_msg_t *msg)
{
    uint8_t *paramStart;
    uint8_t id;
    char *k, *sp;

    /* Database empty */
    if (msg->data[2] == '\n')
    {
        channelInfo dummy;
        signal Database.receivedChannelEntry(0xff, dummy);
```

```
        return;
    }

    /* Invalid message */
    if (!prepareMessage(msg, &paramStart))
        return;

    list.entries = 0;
    list.currId = 0;
    k = strtok_r((char *) paramStart, ",", &sp);

    while (k != NULL)
    {
        id = (uint8_t) strtol(k, NULL, 10);
        if (id < DB_MAX_ENTRIES)
            list.ids[list.entries++] = id;

        k = strtok_r(NULL, ",", &sp);
    }

    if (list.entries > 0)
    {
        getList = TRUE;
        post fetchList();
    }
}

/*
 * @brief       Decode get command response and signal channel.
 * @param msg   The message to decode.
 */
static void decodeGet(udp_msg_t *msg)
{
    uint8_t *paramStart;
    uint8_t id;
    /* Space for NULL termination */
    char name[NAME_MAX_LEN+1];
    char notes[NOTE_MAX_LEN+1];
    channelInfo channel;

    /* Invalid message */
    if (!prepareMessage(msg, &paramStart))
        return;

    memset(name, 0, NAME_MAX_LEN+1);
    memset(notes, 0, NOTE_MAX_LEN+1);
    channel.name = name;
    channel.notes = notes;

    /* Valid response */
    if (parseChannelInfo((char *) paramStart, &channel, &id))
    {
        /* Filter favourites */
        if (favOnly)
        {
            if (channel.quickDial > 0)
                signal Database.receivedChannelEntry(id, channel);
        }
        else
            signal Database.receivedChannelEntry(id, channel);
    }
    /* Invalid response */
    else
    {
        channelInfo dummy;
        signal Database.receivedChannelEntry(0xfe, dummy);
    }
}

/*
```

```
 * @brief               Prepare message for further processing, replace '\r' with '\0'.
 * @param msg           Message to prepare.
 * @param paramStart    Pointer to the start of the parameters.
 * @return              FALSE if messge not terminated with '\n'.
 */
static bool prepareMessage(udp_msg_t *msg, uint8_t **paramStart)
{
    /* Valid message is terminated with newline */
    if (msg->data[msg->len-1] != '\n')
        return FALSE;

    msg->data[msg->len-1] = '\0';

    /* Search start of parameter string */
    *paramStart = (uint8_t *) strchr((char *) msg->data, '\r');
    if (*paramStart != NULL)
    {
        **paramStart = '\0';
        (*paramStart)++;
    }
    else
        *paramStart = &msg->data[msg->len];

    return TRUE;
}

/*
 * @brief           Parse a UDP message into a channelInfo struct.
 * @param params    The parameters to parse.
 * @param channel   The struct to parse into.
 * @param id        The ID of the DB entry is stored here.
 * @return          Return if the params were valid.
 */
static bool parseChannelInfo(char *params, channelInfo *channel, uint8_t *id)
{
    char *k, *v, *sp;
    bool gotName = FALSE;

    k = strtok_r(params, "=", &sp);
    if (k == NULL)
        return FALSE;

    while (k != NULL)
    {
        v = sp;

        if (strncmp_P(k, cmd_id, CMD_ID_LEN) == 0)
        {
            uint8_t _id = (uint8_t) strtoul(v, NULL, 10);
            if (_id >= DB_MAX_ENTRIES)
                return FALSE;

            *id = _id;
        }
        else if (strncmp_P(k, cmd_freq, CMD_FREQ_LEN) == 0)
        {
            uint16_t freq = (uint16_t) strtoul(v, NULL, 10);
            if (freq < 875 || freq > 1080)
                return FALSE;

            channel->frequency = freq;
        }
        else if (strncmp_P(k, cmd_picode, CMD_PICODE_LEN) == 0)
        {
            uint16_t pi_code = (uint16_t) strtoul(v, NULL, 10);
            channel->pi_code = pi_code;
        }
        else if (strncmp_P(k, cmd_qdial, CMD_QDIAL_LEN) == 0)
        {
            uint8_t qdial = (uint8_t) strtoul(v, NULL, 10);
```

```
            if (qdial > 9)
                return FALSE;

            channel->quickDial = qdial;
        }
        else if (strncmp_P(k, cmd_name, CMD_NAME_LEN) == 0)
        {
            snprintf(channel->name, NAME_MAX_LEN, "%-8s", v);
            channel->name[NAME_MAX_LEN] = '\0';

            /* Name is either too long or delimiter is missing */
            sp += 8;
            if (*sp != ',')
                return FALSE;

            sp++;
            gotName = TRUE;
        }
        else if (strncmp_P(k, cmd_note, CMD_NOTE_LEN) == 0)
        {
            if (*v != '\0')
                strncpy(channel->notes, v, NOTE_MAX_LEN);

            channel->name[NOTE_MAX_LEN] = '\0';

            /* According to specification, note is the last parameter */
            return TRUE;
        }

        /* Move to next parameter */
        if (!gotName)
        {
            v = strchr(sp, ',');
            if (v == NULL)
                return FALSE;

            sp = v + 1;
        }

        k = strtok_r(NULL, "=", &sp);
        gotName = FALSE;
    }

    /* This should never be reached */
    return FALSE;
}

/*
 * @brief Load the UDP message parameters from PROGMEM.
 */
static void getParamsPgm(params_t *p)
{
    strncpy_P(p->add, cmd_add, CMD_ADD_LEN+1);
    strncpy_P(p->update, cmd_update, CMD_UPDATE_LEN+1);
    strncpy_P(p->id, cmd_id, CMD_ID_LEN+1);
    strncpy_P(p->freq, cmd_freq, CMD_FREQ_LEN+1);
    strncpy_P(p->picode, cmd_picode, CMD_PICODE_LEN+1);
    strncpy_P(p->qdial, cmd_qdial, CMD_QDIAL_LEN+1);
    strncpy_P(p->name, cmd_name, CMD_NAME_LEN+1);
    strncpy_P(p->note, cmd_note, CMD_NOTE_LEN+1);
}

//////////////////////
/* Events          */
//////////////////////

event void UdpSend.sendDone(error_t error)
{
    if (error != SUCCESS)
        post sendTask();
```

```
    }

    /*
     * @brief Copy received message to buffer.
     */
    event void UdpReceive.received(in_addr_t *srcIp, uint16_t srcPort, uint8_t *data, uint16_t
        ↪ len)
    {
        /* Truncate message if too long */
        if (len > MAX_MSG_LEN)
            len = MAX_MSG_LEN;

        memcpy(msgBuf.recv.data, data, len);
        msgBuf.recv.len = len;

        post recvTask();
    }

    event void Control.startDone(error_t error)
    {

    }

    event void Control.stopDone(error_t error)
    {

    }
}
```

Listing 7: ../Database/commands.h

```
/**
 *
 * @file     commands.h
 * @author   Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date     2019-01-15
 *
 * Database commands saved in PROGMEM.
 *
**/

#ifndef __COMAMNDS__
#define __COMMANDS__

#include <avr/pgmspace.h>

#define CMD_OK_LEN 2
const char cmd_ok[] PROGMEM = "ok";
#define CMD_ADD_LEN 3
const char cmd_add[] PROGMEM = "add";
#define CMD_UPDATE_LEN 6
const char cmd_update[] PROGMEM = "update";
#define CMD_LIST_LEN 4
const char cmd_list[] PROGMEM = "list";
#define CMD_PURGEALL_LEN 8
const char cmd_purgeall[] PROGMEM = "purgeall";
#define CMD_MSG_LEN 3
const char cmd_msg[] PROGMEM = "msg";
#define CMD_GET_LEN 3
const char cmd_get[] PROGMEM = "get";
#define CMD_DBFULL_LEN 15
const char cmd_dbFull[] PROGMEM = "Channel DB Full";
#define CMD_ID_LEN 2
const char cmd_id[] PROGMEM = "id";
#define CMD_FREQ_LEN 4
const char cmd_freq[] PROGMEM = "freq";
#define CMD_PICODE_LEN 6
const char cmd_picode[] PROGMEM = "picode";
#define CMD_QDIAL_LEN 5
```

```
const char cmd_qdial[] PROGMEM = "qdial";
#define CMD_NAME_LEN 4
const char cmd_name[] PROGMEM = "name";
#define CMD_NOTE_LEN 4
const char cmd_note[] PROGMEM = "note";

#endif
```

Listing 8: ../Database/udp_config.h

```
#ifndef UDP_CONFIG_H
#define UDP_CONFIG_H

#define CUSTOM_IP_SETTINGS

#define UDP_PORT     50000UL
// note the ',' (instead of the usual '.') between numbers
#define DESTINATION 192,168,42,1

// the following settings are only applied if CUSTOM_IP_SETTINGS is defined
// note the ',' (instead of the usual '.') between numbers
#define IP  192,168,42,2
#define NETMASK 255,255,255,0
#define GATEWAY 192,168,42,1

// Memory Pool Settings
#define MAX_MSG_LEN     128
#define MSG_POOL_SIZE   128

typedef struct udp_msg {
        uint16_t len;
    uint8_t data[MAX_MSG_LEN];
} udp_msg_t;

#endif
```

A.3 FMClick

Listing 9: ../FMClick/FMClick.nc

```
// A Thomas Lamprecht <tlamprecht@ecs.tuwien.ac> production - 2018

#define PS_BUF_SZ   8
#define RT_BUF_SZ   64
#define CT_BUF_SZ   6

typedef enum {
    UP,
    DOWN,
    BAND
} seekmode_t;

typedef enum {
    PS, // Programm Station
    RT, // Radio Text
    TIME, // TIME
} RDSType;

interface FMClick {
    command error_t tune(uint16_t channel);

    command error_t seek(seekmode_t mode);

    command uint16_t getChannel(void);

    command error_t setVolume(uint8_t);
```

```
    command error_t receiveRDS(bool enable);

    event void initDone(error_t res);

    async event void tuneComplete(uint16_t channel);

    async event void seekComplete(uint16_t channel);

    async event void rdsReceived(RDSType type, char *buf);
}
```

Listing 10: ../FMClick/FMClickC.nc

```
/**
 *
 * @file FMClickC.nc
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-12-12
 *
 * Configuration of the FMClick module.
 *
**/

configuration FMClickC {
    provides
    {
        interface Init;
        interface FMClick;
    };
}

implementation {
    components FMClickP, GlcdC;
    components HplAtm128InterruptC as Int;
    components new Atm128I2CMasterC() as I2C;
    components new TimerMilliC() as Timer;
    components HplAtm1280GeneralIOC as IO;

    Init = FMClickP.Init;
    FMClick = FMClickP.FMClick;

    FMClickP.Int -> Int.Int3;

    FMClickP.I2CResource -> I2C.Resource;
    FMClickP.I2C -> I2C.I2CPacket;

    FMClickP.Timer -> Timer;

    FMClickP.RSTPin -> IO.PortD2;
    FMClickP.SDIOPin -> IO.PortD1;
    FMClickP.INTPin -> IO.PortD3;

    FMClickP.Glcd -> GlcdC.Glcd;
}
```

Listing 11: ../FMClick/FMClickP.nc

```
/**
 *
 * @file FMClickP.nc
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-12-12
 *
 * FMClick module implementation.
 *
**/

#include <stdio.h>
```

```
#include <string.h>

module FMClickP {
    provides
    {
        interface Init;
        interface FMClick;
    }
    uses
    {
        interface HplAtm128Interrupt as Int;
        interface Resource as I2CResource;
        interface I2CPacket<TI2CBasicAddr> as I2C;
        interface Timer<TMilli> as Timer;
        interface GeneralIO as RSTPin;
        interface GeneralIO as SDIOPin;
        interface GeneralIO as INTPin;
        interface Glcd;
    }
}

implementation {
    /* I2C addresses */
    #define DEVICE_WRITE_ADDR   0x10
    #define DEVICE_READ_ADDR    0x10

    /* Register file parameters */
    #define REGISTER_NUM        16
    #define REGISTER_WIDTH      2       /* Bytes per register word */
    #define READ_START_ADDR     0x0a    /* First address read by I2C communication */
    #define WRITE_START_ADDR    0x02    /* First address written by I2C communication */

    /* Register file addresses */
    #define DEVID_REG           0x00
    #define CHIPID_REG          0x01
    #define POWERCONF_REG       0x02
    #define CHANNEL_REG         0x03
    #define SYSCONF1_REG        0x04
    #define SYSCONF2_REG        0x05
    #define SYSCONF3_REG        0x06
    #define TEST1_REG           0x07
    #define STATRSSI_REG        0x0a
    #define READCHAN_REG        0x0b
    #define RDSA_REG            0x0c
    #define RDSB_REG            0x0d
    #define RDSC_REG            0x0e
    #define RDSD_REG            0x0f

    /* Chip ID */
    #define POWERUP_VAL         0x1253
    #define POWERDOWN_VAL       0x1200

    /* Power Configuration */
    #define ENABLE_MASK         0x0001
    #define DISABLE_MASK        0x0040
    #define DMUTE_MASK          0x4000
    #define SEEKUP_MASK         0x0200
    #define SKMODE_MASK         0x0400
    #define SEEK_MASK           0x0100
    #define RDSM_MASK           0x0800

    /* Channel */
    #define TUNE_MASK           0x8000
    #define CHAN_MASK           0x03ff

    /* System Configuration 1 */
    #define GPIO2_MASK          0x000c
    #define GPIO2_VAL           0x0004  /* Configures GPIO2 to fire RDS/STC interrupts */
    #define BLNDADJ_MASK        0x00c0
    #define BLNDADJ_VAL         0x0000  /* Default */
```

```
#define RDS_MASK              0x1000
#define RDSIEN_MASK           0x8000
#define STCIEN_MASK           0x4000
#define DE_MASK               0x0800 /* Europe FM de-emphasis settings */

/* System Configuration 2 */
#define SEEKTH_MASK           0xff00
#define SEEKTH_VAL            0x1900  /* Recommended */
#define BAND_MASK             0x00c0
#define BAND_VAL              0x0000 /* European FM band */
#define SPACE_MASK            0x0030
#define SPACE_VAL             0x0010 /* Europe FM channel spacing */
#define VOLUME_MASK           0x000f

/* System Configuration 3 */
#define VOLEXT_MASK           0x0100
#define SKSNR_MASK            0x00f0
#define SKSNR_VAL             0x0040  /* Good SNR threshold */
#define SKCNT_MASK            0x000f
#define SKCNT_VAL             0x0008  /* More stringent valid station requirements */

/* Test 1 */
#define XOSCEN_MASK           0x8000

/* Status RSSI */
#define RDSR_MASK             0x8000
#define STC_MASK              0x4000
#define SFBL_MASK             0x2000
#define BLERA_MASK            0x0600
#define ST_MASK               0x0100
#define RSSI_MASK             0x00ff

/* Read Channel */
#define READCHAN_MASK         0x03ff

#define RESET_DELAY_MS        1
#define READ_DELAY_MS         100
#define XOSCEN_DELAY_MS       750
#define POWERUP_DELAY_MS      150

#define BAND_LOW_END          875
#define BAND_HIGH_END         1080

uint16_t shadowRegisters[REGISTER_NUM];
uint8_t comBuffer[REGISTER_NUM*REGISTER_WIDTH];
uint8_t writeAddr;

enum driver_state {IDLE, INIT, TUNE, SEEK, RDSEN, VOL, RDS};
enum bus_state {NOOP, READ, WRITE};
enum init_state {SETUP, INITREG, XOSCEN, WAITXOSC, ENABLE, WAITPOWERUP, READREGF, CONFIG,
    ↪ FINISH};
enum tune_state {STARTTUNE, WAITTUNE, TUNECHAN, ENDTUNE, READTUNE, FINTUNE};
enum seek_state {STARTSEEK, WAITSEEK, SEEKCHAN, ENDSEEK, READSEEK, FINSEEK};
enum rds_state {READRDS, DECODERDS};

struct
{
    enum driver_state   driver;
    enum bus_state      bus;
    enum init_state     init;
    enum tune_state     tune;
    enum seek_state     seek;
    enum rds_state      rds;
} states;

bool RDSen;

uint16_t currChannel;
uint16_t nextChannel;
seekmode_t seekMode;
```

```
/* RDS group type codes */
#define GT_0A 0x00
#define GT_0B 0x01
#define GT_2A 0x04
#define GT_2B 0x05
#define GT_4A 0x08

/* RDS blocks per message */
#define PS_BLOCKS    (PS_BUF_SZ/2)
#define RT_BLOCKS    (RT_BUF_SZ/4)

struct
{
    uint8_t PSBlocks;
    char PS[PS_BUF_SZ+2]; /* Two additional bytes for storing the PI code */
    char RT[RT_BUF_SZ];
    char CT[CT_BUF_SZ];
} rds;

/* Task prototypes */
task void init(void);
task void tune(void);
task void seek(void);
task void decodeRDS(void);
task void readRegisters(void);
task void readI2C(void);
task void writeRegisters(void);
task void writeI2C(void);
task void registerWriteback(void);

/* Function prototypes */
static void enableRDS(bool enable);

//////////////////////
/* Interface commands */
//////////////////////

/*
 * @brief   Initialize the FMClick module.
 * @return  If the initialization was started, SUCCESS is returned, else FAIL.
 */
command error_t Init.init(void)
{
    enum driver_state state;
    atomic { state = states.driver; }

    if (IDLE != state)
        return FAIL;

    atomic
    {
        states.driver = INIT;
        states.bus = NOOP;
        states.init = SETUP;
        RDSen = FALSE;
        memset(shadowRegisters, 0, sizeof(shadowRegisters));
        memset(comBuffer, 0, sizeof(comBuffer));
    }

    /* Start board reset */
    call RSTPin.makeOutput();
    call RSTPin.clr();

    /* Select 2-wire mode */
    call SDIOPin.makeOutput();
    call SDIOPin.clr();

    /* Set interrupt pin as input, disable pullup and set falling edge on STC/RDS
        ↪ interrupt */
```

```
    call INTPin.makeInput();
    call INTPin.clr();
    call Int.edge(FALSE);

    call Timer.startOneShot(RESET_DELAY_MS);
    return SUCCESS;
}

/*
 * @brief               Tune to a specific channel.
 * @parameter channel   The radio channel frequency*10 (e.g. for 103.8 pass 1038).
 * @return              If the channel is in the band and the tuning process was
 *                      started, SUCCESS is returned, else FAIL.
 */
command error_t FMClick.tune(uint16_t channel)
{
    enum driver_state state;

    atomic { state = states.driver; }

    if (IDLE != state)
        return FAIL;

    /* Check if channel is in the allowed band */
    if (channel < BAND_LOW_END || channel > BAND_HIGH_END)
        return FAIL;

    atomic
    {
        states.driver = TUNE;
        states.tune = STARTTUNE;
        nextChannel = channel;
    }

    post tune();
    return SUCCESS;
}

/*
 * @brief       Seek channels in the band.
 * @param mode  The desired seek mode (UP to seek the next higher channel,
 *              DOWN to seek the next lower channel and BAND to seek all
 *              channels on the band).
 * @return      If seeking was started, SUCCESS is returned, else FAIL.
 */
command error_t FMClick.seek(seekmode_t mode)
{
    enum driver_state state;
    atomic { state = states.driver; }

    if (IDLE != state)
        return FAIL;

    atomic
    {
        states.driver = SEEK;
        states.seek = STARTSEEK;
        seekMode = mode;
    }

    post seek();
    return SUCCESS;
}

/*
 * @brief   Get the frequency of the channel the module is tuned to.
 * @return  The channel frequency * 10;
 */
command uint16_t FMClick.getChannel(void)
{
```

```
    return currChannel;
}

/*
 * @brief           Set the volume of the sound output on the FMClick module.
 * @param volume    The volume to set (max. 15).
 * @return          If volume setting was started, SUCCESS is returned, else FAIL.
 */
command error_t FMClick.setVolume(uint8_t volume)
{
    enum driver_state state;
    atomic { state = states.driver; }

    if (IDLE != state)
        return FAIL;

    atomic
    {
        states.driver = VOL;
        shadowRegisters[SYSCONF2_REG] = (shadowRegisters[SYSCONF2_REG] & ~VOLUME_MASK) |
                                        (volume & VOLUME_MASK);
        writeAddr = SYSCONF2_REG;
    }
    post writeRegisters();

    return SUCCESS;
}

/*
 * @brief           Set the volume of the sound output on the FMClick module.
 * @param volume    The volume to set (max. 15).
 * @return          If volume setting was started, SUCCESS is returned, else FAIL.
 */
command error_t FMClick.receiveRDS(bool enable)
{
    enum driver_state state;
    bool en;

    atomic
    {
        en = RDSen;
        state = states.driver;
        RDSen = enable;
    }

    if (IDLE == state)
    {
        if (enable != en)
        {
            atomic { states.driver = RDSEN; }
            enableRDS(enable);
        }
    }

    return SUCCESS;
}

//////////////////////
/* Tasks            */
//////////////////////

/*
 * @brief Initialization state machine. Signals initDone().
 */
task void init(void)
{
    enum init_state state;
    atomic { state = states.init; }

    if (SETUP == state)
```

```
{
    /* Finish reset */
    call RSTPin.set();
    call Timer.startOneShot(READ_DELAY_MS);
    atomic { states.init = INITREG; }
}
if (INITREG == state)
{
    /* Read initial register file state */
    atomic { states.init = XOSCEN; }
    post readRegisters();
}
else if (XOSCEN == state)
{
    /* Start internal oscillator and clear RDSD */
    atomic
    {
        shadowRegisters[TEST1_REG] |= XOSCEN_MASK;
        shadowRegisters[RDSD_REG] = 0x0000;
        writeAddr = RDSD_REG;
        states.init = WAITXOSC;
    }
    post writeRegisters();
}
else if (WAITXOSC == state)
{
    /* Wait for oscillator to stabilize */
    atomic { states.init = ENABLE; }
    call Timer.startOneShot(XOSCEN_DELAY_MS);
}
else if (ENABLE == state)
{
    atomic
    {
        /* Start device powerup and disable mute */
        shadowRegisters[POWERCONF_REG] = 0x4001;
        writeAddr = POWERCONF_REG;
        states.init = WAITPOWERUP;
    }
    post writeRegisters();
}
else if (WAITPOWERUP == state)
{
    /* Wait for chip powerup */
    atomic { states.init = READREGF; }
    call Timer.startOneShot(POWERUP_DELAY_MS);
}
else if (READREGF == state)
{
    /* Read register file state */
    atomic { states.init = CONFIG; }
    post readRegisters();
}
else if (CONFIG == state)
{
    uint16_t chipIDReg;
    atomic { chipIDReg = shadowRegisters[CHIPID_REG]; }

    if (POWERUP_VAL != chipIDReg)
    {
        atomic { states.driver = IDLE; }
        signal FMClick.initDone(FAIL);
        return;
    }

    atomic
    {
        /* Enable STC interrupt and configure GPIO2 for interrupt transmission */
        shadowRegisters[SYSCONF1_REG] = (shadowRegisters[SYSCONF1_REG] &
                                         ~(GPIO2_MASK | BLNDADJ_MASK | RDS_MASK |
```

```
                                                    ↪ RDSIEN_MASK)) |
                                                GPIO2_VAL | BLNDADJ_VAL | STCIEN_MASK |
                                                    ↪ DE_MASK;
            /* General and regional configuration */
            shadowRegisters[SYSCONF2_REG] = (shadowRegisters[SYSCONF2_REG] &
                                                ~(SEEKTH_MASK | BAND_MASK | SPACE_MASK |
                                                    ↪ VOLUME_MASK)) |
                                                SEEKTH_VAL | BAND_VAL | SPACE_VAL;
            shadowRegisters[SYSCONF3_REG] = (shadowRegisters[SYSCONF3_REG] &
                                                ~(VOLEXT_MASK | SKSNR_MASK | SKCNT_MASK)) |
                                                SKSNR_VAL | SKCNT_MASK;
            writeAddr = SYSCONF3_REG;
            states.init = FINISH;
        }

        post writeRegisters();
    }
    else if (FINISH == state)
    {
        atomic { states.driver = IDLE; }
        signal FMClick.initDone(SUCCESS);
    }
}

/*
 * @brief Tuning state machine. Signals tuneComplete().
 */
task void tune(void)
{
    enum tune_state state;
    atomic { state = states.tune; }

    if (STARTTUNE == state)
    {
        atomic
        {
            shadowRegisters[CHANNEL_REG] = TUNE_MASK | (CHAN_MASK & (nextChannel -
                ↪ BAND_LOW_END));
            shadowRegisters[SYSCONF1_REG] |= STCIEN_MASK;
            shadowRegisters[SYSCONF1_REG] &= ~(RDS_MASK | RDSIEN_MASK);
            writeAddr = SYSCONF1_REG;
            states.tune = WAITTUNE;
        }
        post writeRegisters();
    }
    else if (WAITTUNE == state)
    {
        /* Enable STC interrupt */
        atomic { states.tune = TUNECHAN; }
        call Int.clear();
        call Int.enable();
    }
    else if (TUNECHAN == state)
    {
        atomic { states.tune = ENDTUNE; }
        post readRegisters();
    }
    else if (ENDTUNE == state)
    {
        /* Read channel and disable tuning */
        atomic
        {
            currChannel = (shadowRegisters[READCHAN_REG] & READCHAN_MASK) + BAND_LOW_END;
            shadowRegisters[CHANNEL_REG] &= ~TUNE_MASK;
            shadowRegisters[SYSCONF1_REG] &= ~STCIEN_MASK;
            writeAddr = SYSCONF1_REG;
            states.tune = READTUNE;
        }
        post writeRegisters();
    }
```

```
    else if (READTUNE == state)
    {
        /* Read registers to check STC bit */
        atomic { states.tune = FINTUNE; }
        post readRegisters();
    }
    else if (FINTUNE == state)
    {
        uint8_t stc;
        atomic { stc = (shadowRegisters[STATRSSI_REG] & STC_MASK) >> 8; }

        /* Tuning complete */
        if (!stc)
        {
            signal FMClick.tuneComplete(currChannel);
            if (RDSen)
            {
                atomic { states.driver = RDSEN; }
                enableRDS(TRUE);
            }
            else
                atomic { states.driver = IDLE; }
        }
        /* Read the register file until STC is cleared */
        else
        {
            atomic { states.tune = READTUNE; }
            post tune();
        }
    }
}

/*
 * @brief Seeking state machine. Signals tuneComplete().
 */
void task seek(void)
{
    enum seek_state state;
    seekmode_t mode;
    static uint8_t sfbl;

    atomic
    {
        mode = seekMode;
        state = states.seek;
    }

    if (STARTSEEK == state)
    {
        /* Start at lower band end and stop at high band end for band seek */
        if (BAND == mode)
            atomic { shadowRegisters[POWERCONF_REG] = shadowRegisters[POWERCONF_REG] |
                ↪ SKMODE_MASK | SEEK_MASK; }
        /* Wrap around band limits for single seek */
        else
            atomic { shadowRegisters[POWERCONF_REG] = (shadowRegisters[POWERCONF_REG] & ~
                ↪ SKMODE_MASK) | SEEK_MASK; }

        if (UP == mode || BAND == mode)
            atomic { shadowRegisters[POWERCONF_REG] |= SEEKUP_MASK; }
        else
            atomic { shadowRegisters[POWERCONF_REG] &= ~SEEKUP_MASK; }

        atomic
        {
            shadowRegisters[SYSCONF1_REG] |= STCIEN_MASK;
            shadowRegisters[SYSCONF1_REG] &= ~(RDS_MASK | RDSIEN_MASK);
            writeAddr = SYSCONF1_REG;
            states.seek = WAITSEEK;
        }
```

```
        post writeRegisters();
    }
    else if (WAITSEEK == state)
    {
        /* Enable STC interrupt */
        atomic { states.seek = SEEKCHAN; }
        call Int.clear();
        call Int.enable();
    }
    else if (SEEKCHAN == state)
    {
        atomic { states.seek = ENDSEEK; }
        post readRegisters();
    }
    else if (ENDSEEK == state)
    {
        /* Read sfbl bit and channel and disable seeking */
        atomic
        {
            sfbl = (uint8_t)((shadowRegisters[STATRSSI_REG] & SFBL_MASK) >> 8);
            currChannel = (shadowRegisters[READCHAN_REG] & READCHAN_MASK) + BAND_LOW_END;
            shadowRegisters[POWERCONF_REG] &= ~SEEK_MASK;
            shadowRegisters[SYSCONF1_REG] &= ~STCIEN_MASK;
            writeAddr = SYSCONF1_REG;
            states.seek = READSEEK;
        }
        post writeRegisters();
    }
    else if (READSEEK == state)
    {
        /* Read registers to check STC bit */
        atomic { states.seek = FINSEEK; }
        post readRegisters();
    }
    else if (FINSEEK == state)
    {
        uint8_t stc;
        atomic { stc = (shadowRegisters[STATRSSI_REG] & STC_MASK) >> 8; }

        /* Seek complete */
        if (!stc)
        {
            /* Channel valid */
            if (!sfbl)
            {
                signal FMClick.seekComplete(currChannel);

                if (RDSen)
                {
                    atomic { states.driver = RDSEN; }
                    enableRDS(TRUE);
                }
                else
                    atomic { states.driver = IDLE; }
            }
            /* Reached band end / no valid channel found */
            else
            {
                signal FMClick.seekComplete(0xffff);

                if (RDSen)
                {
                    atomic { states.driver = RDSEN; }
                    enableRDS(TRUE);
                }
                else
                    atomic { states.driver = IDLE; }
            }
        }
        /* Read the register file until STC is cleared */
```

```
        else
        {
            atomic { states.seek = READSEEK; }
            post seek();
        }
    }
}

/*
 * @brief Decodeing state machine for RDS messages. Signals rdsReceived().
 */
void task decodeRDS(void)
{
    enum rds_state state;
    atomic { state = states.rds; }

    if (READRDS == state)
    {
        atomic { states.rds = DECODERDS; }
        post readRegisters();
    }
    else if (DECODERDS == state)
    {
        uint8_t groupType;
        uint16_t RDSA, RDSB, RDSC, RDSD;
        uint8_t offset, blocks;
        uint8_t hours, minutes, localOffset;

        atomic
        {
            RDSA = shadowRegisters[RDSA_REG];
            RDSB = shadowRegisters[RDSB_REG];
            RDSC = shadowRegisters[RDSC_REG];
            RDSD = shadowRegisters[RDSD_REG];
        }

        groupType = (uint8_t)(RDSB >> 11);

        switch (groupType)
        {
            /* Intended fallthrough, packets get decoded in the same way */
            case GT_0A:
            case GT_0B:
                offset = ((uint8_t)RDSB) & 0x03;
                atomic
                {
                    /* PI code */
                    rds.PS[PS_BUF_SZ] = (char) (RDSA >> 8);
                    rds.PS[PS_BUF_SZ+1] = (char) RDSA;
                    /* Station Name */
                    rds.PS[offset<<1] = (char)(RDSD >> 8);
                    rds.PS[(offset<<1)+1] = (char)RDSD;
                    rds.PSBlocks |= (1<<offset);
                    blocks = rds.PSBlocks;
                }
                if (blocks == 0x0f)
                {
                    signal FMClick.rdsReceived(PS, rds.PS);
                    memset(rds.PS, '␣', PS_BUF_SZ);
                    rds.PSBlocks = 0;
                }
                break;

            case GT_2A:
                offset = ((uint8_t)RDSB) & 0x0f;
                atomic
                {
                    rds.RT[offset<<2] = (char)(RDSC >> 8);
                    rds.RT[(offset<<2)+1] = (char)RDSC;
                    rds.RT[(offset<<2)+2] = (char)(RDSD >> 8);
```

```
                    rds.RT[(offset<<2)+3] = (char)RDSD;
                }
                signal FMClick.rdsReceived(RT, rds.RT);
                break;

            case GT_4A:
                hours = (uint8_t)(RDSD >> 12) | ((uint8_t)(RDSC << 4) & 0x10);
                minutes = ((uint8_t)(RDSD >> 6) & 0x3f);
                localOffset = ((uint8_t)RDSD) & 0x1f;

                /* Determine time offset sign */
                if (RDSD & 0x0020)
                    hours -= localOffset >> 1;
                else
                    hours += localOffset >> 1;

                atomic
                {
                    memset(rds.CT, 0, CT_BUF_SZ);
                    sprintf(rds.CT, "%02d:%02d", hours, minutes);
                }

                signal FMClick.rdsReceived(TIME, rds.CT);
                break;

            default:
                break;
        }

        if (!RDSen)
        {
            atomic
            {
                states.driver = RDSEN;
                states.rds = READRDS;
            }
            enableRDS(FALSE);
        }
        else
        {
            atomic
            {
                states.driver = IDLE;
                states.rds = READRDS;
            }
        }
    }
}

/*
 * @brief Start reading the registers.
 */
task void readRegisters(void)
{
    enum bus_state state;

    /* Make sure the bus is clear */
    atomic { state = states.bus; }
    if (NOOP != state)
    {
        post readRegisters();
        return;
    }

    atomic { states.bus = READ; }
    call I2CResource.request();
}

/*
 * @brief Send read request to the FMClick module.
```

```
 */
task void readI2C(void)
{
    if(call I2C.read(I2C_START | I2C_STOP,
            DEVICE_READ_ADDR,
            REGISTER_NUM*REGISTER_WIDTH,
            comBuffer) != SUCCESS)
    {
        post readI2C();
    }
}

/*
 * @brief Write all registers of the FMClick module up until the address specified in the
    ↪ global writeAddr.
 */
task void writeRegisters(void)
{
    /* Write buffered registers to communication buffer */
    uint8_t i = WRITE_START_ADDR;
    uint8_t j;
    uint8_t bytesToSend;
    enum bus_state state;

    /* Make sure the bus is clear */
    atomic { state = states.bus; }
    if (NOOP != state)
    {
        post writeRegisters();
        return;
    }

    atomic
    {
        states.bus = WRITE;
        bytesToSend = (writeAddr-WRITE_START_ADDR+1)*REGISTER_WIDTH;
    }

    /* Prepare communication buffer for writing */
    for (j = 0; j < bytesToSend; j += REGISTER_WIDTH)
    {
        atomic
        {
            comBuffer[j] = (uint8_t) (shadowRegisters[i] >> 8);
            comBuffer[j+1] = (uint8_t) shadowRegisters[i];
        }
        i = (i+1) & (REGISTER_NUM-1);
    }

    call I2CResource.request();
}

/*
 * @brief Send write request to the FMClick module.
 */
task void writeI2C(void)
{
    uint8_t bytesToSend;
    atomic { bytesToSend = (writeAddr-WRITE_START_ADDR+1)*REGISTER_WIDTH; }

    if (call I2C.write(I2C_START | I2C_STOP,
            DEVICE_WRITE_ADDR,
            bytesToSend,
            comBuffer) != SUCCESS)
    {
        post writeI2C();
    }
}

/*
```

```
 * @brief Write the read register values to the shadow register file in the correct order.
 */
task void registerWriteback(void)
{
    uint8_t i = READ_START_ADDR;
    uint8_t j;
    for (j = 0; j < REGISTER_NUM*REGISTER_WIDTH; j += REGISTER_WIDTH)
    {
        shadowRegisters[i] = (comBuffer[j] << 8) | comBuffer[j+1];
        i = (i+1) & (REGISTER_NUM-1);
    }
}

//////////////////////
/* Internal functions */
//////////////////////

/*
 * @brief           Enable/disable reception of RDS information on the FMCLick board.
 * @param enable    Enable/disable RDS information.
 */
static void enableRDS(bool enable)
{
    if (enable)
    {
        atomic
        {
            shadowRegisters[SYSCONF1_REG] |= (RDS_MASK | RDSIEN_MASK);
            rds.PSBlocks = 0;
            writeAddr = SYSCONF1_REG;
        }

        memset(rds.PS, ' ', PS_BUF_SZ);
        memset(rds.RT, ' ', RT_BUF_SZ);
        memset(rds.CT, ' ', CT_BUF_SZ);

        call Int.clear();
        call Int.enable();
    }
    else
    {
        call Int.disable();
        atomic
        {
            shadowRegisters[SYSCONF1_REG] &= ~(RDS_MASK | RDSIEN_MASK);
            writeAddr = SYSCONF1_REG;
        }
    }

    post writeRegisters();
}

//////////////////////
/* Events */
//////////////////////

/*
 * @brief Reset timer event for init function.
 */
event void Timer.fired()
{
    enum driver_state state;
    atomic { state = states.driver; }

    switch (state)
    {
        case INIT:
            post init();
            break;
```

```
        default:
            break;
    }
}

/*
 * @brief Continue driver operations after successful read.
 */
async event void I2C.readDone(error_t error, uint16_t addr, uint8_t length, uint8_t *data)
{
    enum driver_state state;
    atomic { state = states.driver; }

    if (FAIL == error)
    {
        post readI2C();
        return;
    }

    call I2CResource.release();
    atomic { states.bus = NOOP; }

    post registerWriteback();

    switch (state)
    {
        case INIT:
            post init();
            break;

        case TUNE:
            post tune();
            break;

        case SEEK:
            post seek();
            break;

        case RDS:
            post decodeRDS();
            break;

        default:
            break;
    }
}

/*
 * @brief Continue driver operations after successful write.
 */
async event void I2C.writeDone(error_t error, uint16_t addr, uint8_t length, uint8_t *data
    ↪ )
{
    enum driver_state state;
    atomic { state = states.driver; }

    if (FAIL == error)
    {
        post writeI2C();
        return;
    }

    call I2CResource.release();
    atomic { states.bus = NOOP; }

    switch (state)
    {
        case INIT:
            post init();
            break;
```

```
        case TUNE:
            post tune();
            break;

        case SEEK:
            post seek();
            break;

        /* Intended fallthrough */
        case VOL:
        case RDSEN:
            atomic { states.driver = IDLE; }
            break;

        default:
            break;
    }
}

/*
 * @brief Continue driver operations after RDS/STC interrupt and disable interrupt.
 */
async event void Int.fired()
{
    enum driver_state state;
    atomic { state = states.driver; }

    switch (state)
    {
        case TUNE:
            call Int.disable();
            post tune();
            break;

        case SEEK:
            call Int.disable();
            post seek();
            break;

        case IDLE:
            if (RDSen)
            {
                atomic { states.driver = RDS; }
                post decodeRDS();
            }
            break;

        default:
            break;
    }
}

/*
 * @brief Start read/write when bus is available.
 */
event void I2CResource.granted()
{
    enum driver_state state;
    atomic { state = states.bus; }

    switch (state)
    {
        case READ:
            post readI2C();
            break;

        case WRITE:
            post writeI2C();
            break;
```

```
            default:
                call I2CResource.release();
                break;
        }
    }
}
```

A.4 PS2

Listing 12: ../PS2/PS2.nc

```
interface PS2 {

    // IOs initialisieren, IRQ aktivieren
    command void init(void);

    // character empfangen
    async event void receivedChar(uint8_t chr);

}
```

Listing 13: ../PS2/PS2C.nc

```
/**
 *
 * @file     PS2C.nc
 * @author   Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date     2018-12-09
 *
 * Configuration of the PS2 module.
 *
**/

configuration PS2C {
    provides interface PS2;
}

implementation {
    components PS2P;
    components HplAtmegaPinChange2C as PinChangeIRQ;
    components HplAtm1280GeneralIOC as IO;

    PS2 = PS2P.PS2;
    PS2P.ClockIRQ -> PinChangeIRQ;

    PS2P.ClockPin -> IO.PortK7;
    PS2P.DataPin -> IO.PortK6;
}
```

Listing 14: ../PS2/PS2P.nc

```
/**
 *
 * @file     PS2P.nc
 * @author   Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date     2018-12-09
 *
 * PS2 module implementation.
 *
**/

#include <avr/io.h>
#include <avr/pgmspace.h>
#include <scancodes.h>
```

```
module PS2P {
    provides interface PS2;
    uses
    {
        interface HplAtmegaPinChange as ClockIRQ;
        interface GeneralIO as ClockPin;
        interface GeneralIO as DataPin;
    }
}

implementation {
    #define PS2_BIT_NUM      11
    #define PS2_DATA_START   2
    #define PS2_START_BYTE   0xE0

    #define KB_LEFT_SHIFT    0x12
    #define KB_RIGHT_SHIFT   0x59
    #define KB_UP_KEY        0xF0

    enum kb_shift_state {UNSHIFTED, SHIFTED};
    enum kb_key_state {DOWN, UP};

    static uint8_t PS2BitCount;
    static enum kb_shift_state kbShiftState;
    static enum kb_key_state kbKeyState;

    /*
     * @brief Initialize the pins and pin change interrupt needed
     *        by the PS2 module.
     */
    command void PS2.init(void)
    {
        uint8_t tmpMask;

        /* Configure data & clock pins as input */
        call ClockPin.makeInput();
        call DataPin.makeInput();

        /* Enable PCINT17 (PK1) on the clock line */
        tmpMask = call ClockIRQ.getMask();
        tmpMask |= 1 << PCINT23;
        call ClockIRQ.setMask(tmpMask);

        atomic
        {
            PS2BitCount = PS2_BIT_NUM;
            kbShiftState = UNSHIFTED;
            kbKeyState = DOWN;
        }

        call ClockIRQ.enable();
    }

    /*
     * @brief           This procedure converts a given scancode to a characer.
     *                  If the scancode was converted successfully, it fires a
     *                  signal passing the character. Lower-/uppercase conversion
     *                  is also handled in this procedure.
     * @param scancode  The scancode to be decoded.
     */
    static void decodeScancode(uint8_t scancode)
    {
        if (DOWN == kbKeyState)
        {
            switch (scancode)
            {
                case KB_UP_KEY:
                    kbKeyState = UP;
                    break;
```

```
            case KB_LEFT_SHIFT:
                kbShiftState = SHIFTED;
                break;

            case KB_RIGHT_SHIFT:
                kbShiftState = SHIFTED;
                break;

            /* Perform a scancode-character conversion using a lookup table */
            default:
                if (UNSHIFTED == kbShiftState)
                {
                    uint8_t min = 0;
                    uint8_t max = SC_UNSHIFTED_LEN-1;
                    uint8_t i, sc;

                    /* Perform table lookup with binary search */
                    while (min <= max)
                    {
                        i = (max + min) >> 1;
                        sc = pgm_read_byte(&unshifted[i][0]);

                        if (sc < scancode)
                            min = i+1;
                        else if (sc > scancode)
                            max = i-1;
                        else
                        {
                            /* Fire the signal if the scancode was decoded successfully */
                            signal PS2.receivedChar(pgm_read_byte(&unshifted[i][1]));
                            break;
                        }
                    }
                }
                else
                {
                    uint8_t min = 0;
                    uint8_t max = SC_SHIFTED_LEN-1;
                    uint8_t i, sc;

                    /* Perform table lookup with binary search */
                    while (min <= max)
                    {
                        i = (max + min) >> 1;
                        sc = pgm_read_byte(&shifted[i][0]);

                        if (sc < scancode)
                            min = i+1;
                        else if (sc > scancode)
                            max = i-1;
                        else
                        {
                            /* Fire the signal if the scancode was decoded successfully */
                            signal PS2.receivedChar(pgm_read_byte(&shifted[i][1]));
                            break;
                        }
                    }
                }

        }
    }
    else
    {
        /* Key cannot be released twice in a row */
        kbKeyState = DOWN;

        /* Check if shift keys have been released */
        switch (scancode)
        {
```

```
                case KB_LEFT_SHIFT:
                    kbShiftState = UNSHIFTED;
                    break;

                case KB_RIGHT_SHIFT:
                    kbShiftState = UNSHIFTED;
                    break;

                default:
                    break;
            }
        }
    }

    /*
     * @brief Capture an edge change on the clock line. Sample data bits
     *        on falling edge and pass a data byte to the scancode-decoder.
     */
    async event void ClockIRQ.fired()
    {
        static uint8_t PS2Data;

        /* Falling clock edge captured */
        if (!(call ClockPin.get()))
        {
            /* Only sample data bits */
            if (PS2BitCount < PS2_BIT_NUM && PS2BitCount > PS2_DATA_START)
            {
                PS2Data = PS2Data >> 1;
                if (call DataPin.get())
                    PS2Data |= 0x80;
            }
        }
        /* Rising edge captured */
        else
        {
            PS2BitCount--;

            /* Received a whole data byte */
            if (PS2BitCount == 0)
            {
                /* Ignore start byte */
                if (PS2Data != PS2_START_BYTE)
                    decodeScancode(PS2Data);

                PS2BitCount = PS2_BIT_NUM;
            }
        }
    }
}
```

Listing 15: ../PS2/scancodes.h

```
#ifndef SCANCODES_H
#define SCANCODES_H

// Unshifted characters
#define SC_UNSHIFTED_LEN 66
uint8_t const PROGMEM unshifted[][2] = {
{0x0d,9},
{0x0e,'|'},
{0x15,'q'},
{0x16,'1'},
{0x1a,'y'},
{0x1b,'s'},
{0x1c,'a'},
{0x1d,'w'},
{0x1e,'2'},
{0x21,'c'},
```

```
{0x22,'x'},
{0x23,'d'},
{0x24,'e'},
{0x25,'4'},
{0x26,'3'},
{0x29,' '},
{0x2a,'v'},
{0x2b,'f'},
{0x2c,'t'},
{0x2d,'r'},
{0x2e,'5'},
{0x31,'n'},
{0x32,'b'},
{0x33,'h'},
{0x34,'g'},
{0x35,'z'},
{0x36,'6'},
{0x39,','},
{0x3a,'m'},
{0x3b,'j'},
{0x3c,'u'},
{0x3d,'7'},
{0x3e,'8'},
{0x41,','},
{0x42,'k'},
{0x43,'i'},
{0x44,'o'},
{0x45,'0'},
{0x46,'9'},
{0x49,'.'},
{0x4a,'-'},
{0x4b,'l'},
{0x4c,'o'},
{0x4d,'p'},
{0x4e,'+'},
{0x52,'o'},
{0x54,'o'},
{0x55,'\\'},
{0x5a,'\n'},
{0x5b,'x'},
{0x5d,'\''},
{0x61,'<'},
{0x66,'\b'},
{0x69,'1'},
{0x6b,'4'},
{0x6c,'7'},
{0x70,'0'},
{0x71,','},
{0x72,'2'},
{0x73,'5'},
{0x74,'6'},
{0x75,'8'},
{0x79,'+'},
{0x7a,'3'},
{0x7b,'-'},
{0x7c,'*'},
{0x7d,'9'},
{0,0}
};
// Shifted characters
#define SC_SHIFTED_LEN 66
uint8_t const PROGMEM shifted[][2] = {
{0x0d,9},
{0x0e,'x'},
{0x15,'Q'},
{0x16,'!'},
{0x1a,'Y'},
{0x1b,'S'},
{0x1c,'A'},
{0x1d,'W'},
```

```
{0x1e,'"'},
{0x21,'C'},
{0x22,'X'},
{0x23,'D'},
{0x24,'E'},
{0x25,'x'},
{0x26,'#'},
{0x29,'␣'},
{0x2a,'V'},
{0x2b,'F'},
{0x2c,'T'},
{0x2d,'R'},
{0x2e,'%'},
{0x31,'N'},
{0x32,'B'},
{0x33,'H'},
{0x34,'G'},
{0x35,'Z'},
{0x36,'&'},
{0x39,'L'},
{0x3a,'M'},
{0x3b,'J'},
{0x3c,'U'},
{0x3d,'/'},
{0x3e,'('},
{0x41,';'},
{0x42,'K'},
{0x43,'I'},
{0x44,'O'},
{0x45,'='},
{0x46,')'},
{0x49,':'},
{0x4a,'_'},
{0x4b,'L'},
{0x4c,'x'},
{0x4d,'P'},
{0x4e,'?'},
{0x52,'x'},
{0x54,'x'},
{0x55,'`'},
{0x5a,'\n'},
{0x5b,'^'},
{0x5d,'*'},
{0x61,'>'},
{0x66,'\b'},
{0x69,'1'},
{0x6b,'4'},
{0x6c,'7'},
{0x70,'0'},
{0x71,','},
{0x72,'2'},
{0x73,'5'},
{0x74,'6'},
{0x75,'8'},
{0x79,'+'},
{0x7a,'3'},
{0x7b,'-'},
{0x7c,'*'},
{0x7d,'9'},
{0,0}
};


#endif
```

A.5 VolumeAdc

Listing 16: ../VolumeAdc/VolumeAdcC.nc

```
/**
 *
 * @file VolumeAdcC.nc
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-12-09
 *
 * Configuration of the VolumeAdc module.
 *
**/

#include <debug.h>

configuration VolumeAdcC {
    provides interface Read<uint16_t>;
}

implementation {
    components VolumeAdcP;
    components new AdcReadClientC() as Adc;
    components HplAtm1280GeneralIOC as IO;

    Read = Adc.Read;

    VolumeAdcP.VolumePin -> IO.PortF0;

    VolumeAdcP.Atm1280AdcConfig <- Adc.Atm1280AdcConfig;
    VolumeAdcP.ResourceConfigure <- Adc.ResourceConfigure;
}
```

Listing 17: ../VolumeAdc/VolumeAdcP.nc

```
/**
 *
 * @file VolumeAdcP.nc
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-12-09
 *
 * VolumeAdc module implementation.
 *
**/

#include <debug.h>

module VolumeAdcP {
    provides
    {
        interface Atm1280AdcConfig;
        interface ResourceConfigure;
    };
    uses interface GeneralIO as VolumePin;
}

implementation {
    async command uint8_t Atm1280AdcConfig.getChannel()
    {
        return ATM1280_ADC_SNGL_ADC0;
    }

    async command uint8_t Atm1280AdcConfig.getRefVoltage()
    {
        return ATM1280_ADC_VREF_AVCC;
    }

    async command uint8_t Atm1280AdcConfig.getPrescaler()
    {
        return ATM1280_ADC_PRESCALE_128;
    }
```

```
    async command void ResourceConfigure.configure()
    {
        call VolumePin.makeInput();
    }

    async command void ResourceConfigure.unconfigure()
    {

    }
}
```

A.6 Network Stack

Listing 18: ../PingP.nc

```
/**
 * @author: Andreas Hagmann <ahagmann@ecs.tuwien.ac.at>
 * @date:   12.12.2011
 *
 * based on an implementation of Harald Glanzer, 0727156 TU Wien
 */

#include <string.h>

#define ICMP_DATA_LENGTH            100

module PingP {
    uses interface IcmpReceive;
    uses interface IcmpSend;
}

implementation {

        /* Received echo request - just send the packet back */
    event void IcmpReceive.received(in_addr_t *srcIp, uint8_t code, uint8_t *data, uint16_t
        ↪ len) {
            in_addr_t destIp;
            uint8_t dataBuf[ICMP_DATA_LENGTH];

            memcpy(&destIp, srcIp, sizeof(in_addr_t));
            memset(&dataBuf, 0, ICMP_DATA_LENGTH);
            memcpy(&dataBuf, data, ICMP_DATA_LENGTH);

            call IcmpSend.send(&destIp, ICMP_TYPE_ECHO_REPLY, code, dataBuf, len);
    }

    event void IcmpSend.sendDone(error_t error) {

    }
}
```

Listing 19: ../UdpTransceiverP.nc

```
/**
 * @author: Andreas Hagmann <ahagmann@ecs.tuwien.ac.at>
 * @date:   12.12.2011
 *
 * based on an implementation of Harald Glanzer, 0727156 TU Wien
 */

#include "udp.h"

#define ICMP_TYPE_PORT_UNREACHABLE 3
#define ICMP_CODE_PORT_UNREACHABLE 3
#define ICMP_DATA_LENGTH           100
```

```
module UdpTransceiverP {
    provides interface PacketSender<udp_queue_item_t>;
    provides interface UdpReceive[uint16_t port];
    uses interface IpSend;
    uses interface IpReceive;
    uses interface IcmpSend;
    uses interface IpPacket;
}

implementation {

    udp_packet_t packet;

    command error_t PacketSender.send(udp_queue_item_t *item) {
        // create udp packet

        packet.header.srcPort = item->srcPort;
        packet.header.dstPort = item->dstPort;
        packet.header.len = item->dataLen + sizeof(udp_header_t);
        memcpy(&(packet.data), item->data, item->dataLen);

        return call IpSend.send(&(item->dstIp), (uint8_t*)&(packet), packet.header.len);
    }

        /* default event handler if we do not know what to do with this UDP packet */
    default event void UdpReceive.received[uint16_t port](in_addr_t *srcIp, uint16_t srcPort,
        ↪ uint8_t *data, uint16_t len) {
            in_addr_t destIp;
            uint8_t dataBuf[ICMP_DATA_LENGTH];

            memcpy(&destIp, srcIp, sizeof(in_addr_t));
            memset(&dataBuf, 0, ICMP_DATA_LENGTH);
            memcpy(&dataBuf, data, ICMP_DATA_LENGTH);

            call IcmpSend.send(&destIp, ICMP_TYPE_PORT_UNREACHABLE, ICMP_CODE_PORT_UNREACHABLE
                ↪ , dataBuf, len);
    }

    event void IcmpSend.sendDone(error_t error) {

    }

    event void IpReceive.received(in_addr_t *srcIp, uint8_t *data, uint16_t len) {
        udp_packet_t *p = (udp_packet_t*)data;

        signal UdpReceive.received[p->header.dstPort](srcIp, p->header.srcPort, (uint8_t*)&(p
            ↪ ->data), p->header.len - sizeof(udp_header_t));
    }

    event void IpSend.sendDone(error_t error) {
        signal PacketSender.sendDone(error);
    }
}
```