Élaboration d'un outil de suivi de l'état sanitaire de vaches laitières en pâture par thermographie

Auteur : Sacré, Lucas
Promoteur(s) : Lebeau, Frédéric; Plum, Justine
Faculté : Gembloux Agro-Bio Tech (GxABT)
Diplôme : Master en bioingénieur : sciences et technologies de l'environnement, à finalité spécialisée
Année académique : 2019-2020
URI/URL : http://hdl.handle.net/2268.2/10771

6 ANNEXES

6.1 ANNEXE 1 : CODE D'INITIALISATION

Vsync_app.c

```
#include <stdio.h>
#include <stdlib.h>

#include "FLIR_I2C.h"

#include "LEPTON_OEM.h"
#include "LEPTON_SDK.h"
#include "LEPTON_SYS.h"
#include "LEPTON_Types.h"

#include "LEPTON_RAD.h"



LEP_CAMERA_PORT_DESC_T lepton_port;


//ouverture de la communication i2c, initialisation de la synchronisation
vsync

void init_vsync()
{
    LEP_RESULT result;
    LEP_OEM_GPIO_MODE_E gpio_mode;

    gpio_mode = LEP_OEM_END_GPIO_MODE;
    result = LEP_GetOemGpioMode(&lepton_port, &gpio_mode);
    printf("LEP_GetOemGpioMode gpio_mode = %d result = %d.\n", gpio_mode,
result);

    result = LEP_SetOemGpioMode(&lepton_port, LEP_OEM_GPIO_MODE_VSYNC);
    printf("LEP_SetOemGpioMode result = %d.\n", result);

    gpio_mode = LEP_OEM_END_GPIO_MODE;
    result = LEP_GetOemGpioMode(&lepton_port, &gpio_mode);
    printf("LEP_GetOemGpioMode gpio_mode = %d result = %d.\n", gpio_mode,
result);
}

// activation du mode radiométrique

void enable_rad()
{
    LEP_RESULT result;
    LEP_RAD_ENABLE_E lepton_e;

    lepton_e = LEP_END_RAD_ENABLE;
    result = LEP_GetRadEnableState(&lepton_port, &lepton_e) ;
    printf("LEP_GetRadEnableState = %d result = %d.\n", lepton_e, result);

    result = LEP_SetRadEnableState(&lepton_port, LEP_RAD_ENABLE);
```

```
    printf("LEP_SetRadEnableState result = %d.\n", result);

    lepton_e = LEP_END_RAD_ENABLE;
    result = LEP_GetRadEnableState(&lepton_port, &lepton_e) ;
    printf("LEP_GetRadEnableState = %d result = %d.\n", lepton_e, result);

}

// activation de la correction "Tlinear"
void enable_t_lin()
{
    LEP_RESULT result;
    LEP_RAD_ENABLE_E lepton_e;

    lepton_e = LEP_END_RAD_ENABLE;
    result = LEP_GetRadTLinearEnableState(&lepton_port, &lepton_e) ;
    printf("LEP_GetRadTLinearEnableState = %d result = %d.\n", lepton_e,
result);

    result = LEP_SetRadTLinearEnableState(&lepton_port, LEP_RAD_ENABLE);
    printf("LEP_SetRadTLinearEnableState result = %d.\n", result);

    lepton_e = LEP_END_RAD_ENABLE;
    result = LEP_GetRadTLinearEnableState(&lepton_port, &lepton_e) ;
    printf(" EP_GetRadTLinearEnableState= %d result = %d.\n", lepton_e,
result);

}

// paramétrage de la fonction de transfert Tlinear

void t_lin_parametrisation()
{
    LEP_RESULT result;
    LEP_RAD_FLUX_LINEAR_PARAMS_T t_lin_param;


    result = LEP_GetRadFluxLinearParams(&lepton_port, &t_lin_param) ;
    printf("LEP_GetRadFluxLinearParams = %d result = %d.\n",
t_lin_param.sceneEmissivity, result);

    t_lin_param.sceneEmissivity = 8192*0.98; // emissivité = valeur max *
0.98 (soit em =0.98)
    result = LEP_SetRadFluxLinearParams(&lepton_port, t_lin_param);
    printf(" LEP_SetRadFluxLinearParams result = %d.\n", result);


    result = LEP_GetRadFluxLinearParams(&lepton_port, &t_lin_param) ;
    printf("LEP_GetRadFluxLinearParams = %d result = %d.\n",
t_lin_param.sceneEmissivity, result);


}

// paramétrer la résolution de la valeur des pixels ( pixel = T°K *100)

void t_lin_resolution()
{
    LEP_RESULT result;
    LEP_RAD_TLINEAR_RESOLUTION_E t_lin_resol;
```

```
    t_lin_resol = LEP_RAD_END_RESOLUTION;
    result = LEP_GetRadTLinearResolution(&lepton_port, &t_lin_resol) ;
    printf("LEP_GetRadTLinearResolution = %d result = %d.\n", t_lin_resol,
result);

    result = LEP_SetRadTLinearResolution(&lepton_port,
LEP_RAD_RESOLUTION_0_01);
    printf("LEP_SetRadTLinearResolution result = %d.\n", result);

    t_lin_resol = LEP_RAD_RESOLUTION_0_01;
    result = LEP_GetRadTLinearResolution(&lepton_port, &t_lin_resol) ;
    printf("LEP_GetRadTLinearResolution = %d result = %d.\n", t_lin_resol,
result);


}

// activation du mode "telemetry" et positionnement des metadatas
void config_telemetry()

{
    LEP_RESULT result;
    LEP_SYS_TELEMETRY_ENABLE_STATE_E lepton_tel;

    lepton_tel = LEP_END_TELEMETRY_ENABLE_STATE;
    result = LEP_GetSysTelemetryEnableState(&lepton_port, &lepton_tel) ;
    printf("LEP_GetSysTelemetryEnableState = %d result = %d.\n",
lepton_tel, result);

    result = LEP_SetSysTelemetryEnableState(&lepton_port,
LEP_TELEMETRY_ENABLED);
    printf("LEP_SetSysTelemetryEnableState result = %d.\n", result);

    result = LEP_GetSysTelemetryEnableState(&lepton_port, &lepton_tel) ;
    printf("LEP_GetSysTelemetryEnableState = %d result = %d.\n",
lepton_tel, result);

}

void set_roi()
//encodage des coordonnées du ROI
{
    LEP_RESULT result;
    LEP_RAD_ROI_T roi_param;


    result = LEP_GetRadSpotmeterRoi(&lepton_port, &roi_param) ;


    roi_param.startRow = 1;
    roi_param.startCol = 79;
    roi_param.endRow = 59;
    roi_param.endCol = 159;

    result = LEP_SetRadSpotmeterRoi(&lepton_port, roi_param);

    printf(" LEP_SetRadSpotmeterRoi result = %d.\n", result);
    result = LEP_GetRadSpotmeterRoi(&lepton_port, &roi_param) ;
```

```
    printf(" LEP_GetRadSpotmeterRoi= %d result = %d.\n",roi_param.endRow ,
result);

}


// application d'une correction ffc (flat field correction) pour corriger
la dérive

void lepton_perform_ffc()
{
    LEP_RunSysFFCNormalization(&lepton_port);
}


int main(int argc, char **argv)
{
    LEP_OpenPort(1, LEP_CCI_TWI, 400, &lepton_port);

    enable_rad();
    enable_t_lin();
    //config_telemetry();
    t_lin_resolution();
    t_lin_parametrisation();
    //lepton_perform_ffc();
    set_roi();
    init_vsync();
    return 0;
}
```

6.2 ANNEXE 2 : CODE PRINCIPAL

Principal.py

```
import numpy as np
import struct
from subprocess import Popen
import subprocess
import shlex
import os

from datetime import datetime
import time
from time import sleep
from fct import RFID, Lecture

nbr = 15 #nombre de cliché par série

def lireFichier (emplacement) :
    fichTemp = open(emplacement)
    contenu = fichTemp.read()
    fichTemp.close()
    return contenu

def recupTemp (contenuFich) :
    secondeLigne = contenuFich.split("\n")[1]
    temperatureData = secondeLigne.split(" ")[9]
    temperature = float(temperatureData[2:])
    temperature = temperature / 1000
    return temperature

def extract_roi (): #exécute ROI_process pour extraire les valeurs de t°
max et moyenne de la cible de référence (-->ROI.txt)
    os.system("./ROI_process")

def acquisition_A320():

    command_line=("python3 flir_a320_acquisition_exp3.py")
    args = shlex.split(command_line)
    p1=subprocess.Popen(args)


def acquisition_lepton (): #exécute lepton_data_collector pour enregistrer
une série de images
    tmc = time.time()
    tmc=tmc+(60*60*2)
    file_name = datetime.now()


    os.system("./lepton_data_collector -3 -c %s -o
/home/pi/RaspberryPi_v3/road_step_exe/outputs/frame_"%nbr)

    return tmc,file_name

def process_image (image_file,x,delta,file_name,nbr):

    f = None
```

```
    try:
        f = open(image_file, "rb")
    except IOError:
        timeout = 1
        end = x-1

    else:
        timeout=0
        end=nbr
        n=120*160
        vector = np.zeros((n))

        for i in range(0,n):
            bytes = f.read(2)
            pix_data=struct.unpack('>H',bytes)[0]

            vector[i] = pix_data

        matrix=np.reshape(vector,(120,160))
        array_temp=(matrix/100)-273.15 #+ delta

np.savetxt('/home/pi/RaspberryPi_v3/road_step_exe/outputs/%smat%s.txt'
%(file_name,x), array_temp, fmt='%1.2f')

    finally:
        if f: f.close()

    return timeout,end



def calibration(): # calcul de la différence entre la température moyenne
du ROI et la température de la surface de référence(ds18b20)
    # addition de ce paramètre aux matrices de températures pour les
convertir en températures corrigées
    f = open('/home/pi/RaspberryPi_v3/road_step_exe/outputs/ROI.txt',"r")
    ROI = f.readline()
    ROI = float(ROI)
    ROI_mean =((ROI/100)-273.15)/0.98 #conversion en °C et modification de
l'émissivité supposée à 1 pour la surface de référence
    f.close()

    contenuFich1 = lireFichier("/sys/bus/w1/devices/28-
0000051c2a14/w1_slave")
    temp_S_ref = recupTemp (contenuFich1)

    delta = temp_S_ref - ROI_mean

    return delta

def
Tmean_extraction(delta,temp_S_ref,temp_amb,temp_cible,tmc,file_name,nbr): #
fonction à utiliser en cas de cible fixe (exp 1 et 2)
    data = np.zeros((9))

    f =
open('/home/pi/RaspberryPi_v3/road_step_exe/outputs/summary.txt',"a")
    f2 =
open('/home/pi/RaspberryPi_v3/road_step_exe/outputs/summary_mean.txt',"a")
```

```
     # encoder manuellement la position du centre du bécher pour les
expérimentations 1 et 2
    x = 78
    y = 91

    if nbr > 0 :
        Tm = np.zeros((nbr))
        for k in range (0,nbr):

            array_temp =
np.loadtxt('/home/pi/RaspberryPi_v3/road_step_exe/outputs/%smat%s.txt'
%(file_name,k))
            pix = np.zeros((9))


            pix[0] = array_temp[y-1,x-1]
            pix[1] = array_temp[y-1,x]
            pix[2] = array_temp[y-1,x+1]
            pix[3] = array_temp[y,x-1]
            pix[4] = array_temp[y,x]
            pix[5] = array_temp[y,x+1]
            pix[6] = array_temp[y+1,x-1]
            pix[7] = array_temp[y+1,x]
            pix[8] = array_temp[y+1,x+1]
            Tm[k] =(sum(pix))/9

             #sauvegarde des données finales
            data[0] = tmc  # timecode de la sequence
            data[1] = k #n° de frame
            data[2] = x #coordonnée X pixel max
            data[3] = y # coordonnée Y pixel max
            data[4] = Tm[k]#temperature moyenne
            data[5] = temp_cible #sonde immergée
            data[6] = temp_S_ref #sonde surface de réference
            data[7] = temp_amb #sonde temp ambiante
            data[8] = delta #coeff de calibration de la séquence

            np.savetxt(f, data, fmt='%1.3f', newline=", ")
            f.write("\n")
        f.close()

        # fichier de températures moyennes

        data[4] = sum(Tm)/nbr# moyenne des temperatures moyennes
        data[1] = nbr
        np.savetxt(f2, data, fmt='%1.3f', newline=", ")
        f2.write("\n")
        f2.close()
    else :

          #sauvegarde des données finales
        data[0] = tmc  # timecode de la sequence
        data[1] = 0 #n° de frame
        data[2] = x #coordonnée X pixel max
        data[3] = y # coordonnée Y pixel max
        data[4] = 0
        data[5] = temp_cible #sonde immergée
        data[6] = temp_S_ref #sonde surface de réference
        data[7] = temp_amb #sonde temp ambiante
        data[8] = delta #coeff de calibration de la séquence
```

```
        np.savetxt(f, data, fmt='%1.3f', newline=", ")
        f.write("\n")
        np.savetxt(f2, data, fmt='%1.3f', newline=", ")
        f2.write("\n")
        f.close()
        f2.close()

def
Tmax_extraction(delta,temp_S_ref,temp_amb,temp_cible,tmc,file_name,nbr,num_
genisse): # fonction à utiliser en cas de cible en mouvement
(expérimentation 3 )

    data = np.zeros((10))

    f =
open('/home/pi/RaspberryPi_v3/road_step_exe/outputs/summary.txt',"a")
    f2 =
open('/home/pi/RaspberryPi_v3/road_step_exe/outputs/summary_mean.txt',"a")

    if nbr > 0 :
        M = np.zeros((nbr))
        Tmm = np.zeros((nbr))
        for k in range (0,nbr):

            array_temp =
np.loadtxt('/home/pi/RaspberryPi_v3/road_step_exe/outputs/%smat%s.txt'
%(file_name,k))

            pix = np.zeros((9))

            #classer les valeurs de la matrice sous forme de liste
décroissante
            liste = np.reshape(array_temp,(19200,1))
            liste = liste.tolist()
            liste.sort(reverse=True)


            M[k]=liste[0][0]
            rows, cols = np.where(array_temp==M[k])

            i = rows[0]
            j = cols[0]
            z=0
            while (i == 120 or j == 160): #eviter le cas où le pixel max
est en bordure de frame
                z=z+1
                M[k] = liste[z][0]
                rows, cols = np.where(array_temp==M[k])
                i=rows[0]
                j=cols[0]


            pix[0] = array_temp[i-1,j-1]
            pix[1] = array_temp[i-1,j]
            pix[2] = array_temp[i-1,j+1]
            pix[3] = array_temp[i,j-1]
            pix[4] = array_temp[i,j]
            pix[5] = array_temp[i,j+1]
            pix[6] = array_temp[i+1,j-1]
            pix[7] = array_temp[i+1,j]
```

```
            pix[8] = array_temp[i+1,j+1]

            test=abs(pix-M[k])

            condition=len(np.where(test>2))

            if condition<=4 : # éviter les valeur max isolées
                Tmm[k] =(sum(pix))/9
            else :
                Tmm[k] =999

            #sauvegarde des données finales
            data[0] = tmc  # timecode de la sequence
            #data[1] = int(num_genisse)
            data[2] = k #n° de frame
            data[3] = i #coordonnée X pixel max
            data[4] = j# coordonnée Y pixel max
            data[5] = Tmm[k]#temperature moyenne
            data[6] = temp_cible #sonde immergée
            data[7] = temp_S_ref #sonde surface de réference
            data[8] = temp_amb #sonde temp ambiante
            data[9] = delta #coeff de calibration de la séquence


            np.savetxt(f, data, fmt='%1.3f', newline=", ")
            f.write("\n")

        f.close()
        # fichier de températures moyennes

        data[5] = sum(Tmm)/nbr# moyenne des temperatures moyennes
        data[2] = nbr
        np.savetxt(f2, data, fmt='%1.3f', newline=", ")
        f2.write("\n")
        f2.close()

    else :
          #sauvegarde des données finales
        data[0] = tmc  # timecode de la sequence
        #data[1] = num_genisse
        data[2] = 0 #n° de frame
        data[3] = 0 #coordonnée X pixel max
        data[4] = 0# coordonnée Y pixel max
        data[5] = 0#temperature moyenne
        data[6] = temp_cible #sonde immergée
        data[7] = temp_S_ref #sonde surface de réference
        data[8] = temp_amb #sonde temp ambiante
        data[9] = delta #coeff de calibration de la séquence
        np.savetxt(f, data, fmt='%1.3f', newline=", ")
        f.write("\n")
        np.savetxt(f2, data, fmt='%1.3f', newline=", ")
        f2.write("\n")
        f.close()
        f2.close()

def sequence (num_genisse):
    nbr=25
    timeout=0
    extract_roi ()
    delta = calibration()
```

```
    acquisition_A320()
    tmc,file_name = acquisition_lepton ()
    file_name =file_name.strftime("%d-%m-%H-%M-%S-")
    contenuFich1 = lireFichier("/sys/bus/w1/devices/28-
0000051c2a14/w1_slave")
    contenuFich2 = lireFichier("/sys/bus/w1/devices/28-
0000051beed7/w1_slave")
    contenuFich3 = lireFichier("/sys/bus/w1/devices/28-
0000051c1dff/w1_slave")
    temp_S_ref = recupTemp (contenuFich1)
    temp_cible = recupTemp (contenuFich2)
    temp_amb = recupTemp (contenuFich3)
    ds18b20 = [temp_S_ref,temp_amb,temp_cible]


    for x in range (0,nbr+1):
        if timeout == 0 :

        #boucle pour traiter toutes les acquistions de la série
            if x < 10:

image_file=("/home/pi/RaspberryPi_v3/road_step_exe/outputs/frame_00000%s.gr
ay" %x)
                timeout,end = process_image
(image_file,x,delta,file_name,nbr)
            elif x>9:

image_file=("/home/pi/RaspberryPi_v3/road_step_exe/outputs/frame_0000%s.gra
y" %x)
                timeout,end = process_image
(image_file,x,delta,file_name,nbr)

        else:
            nbr=end
            break
  #supprimer les fichiers .gray
    for m in range (0,end+1):
        if m < 10:

os.remove("/home/pi/RaspberryPi_v3/road_step_exe/outputs/frame_00000%s.gray
" %m)
        elif m>9:

os.remove("/home/pi/RaspberryPi_v3/road_step_exe/outputs/frame_0000%s.gray"
%m)

   # extraction de température


Tmax_extraction(delta,temp_S_ref,temp_amb,temp_cible,tmc,file_name,nbr,num_
genisse)



def main_fonction():
    path_base=
"/home/pi/RaspberryPi_v3/road_step_exe/outputs/identification.txt"
    num_genisse=[]

    ####initialisation RFID
```

```
    reader=RFID()
    #Recherche du port du lecteur RFID
    result=reader.SearchPort()

    while 1:

        #Lecture en continu jusqu'à la lecture d'une boucle
        Lec_num_genisse=Lecture(result) # création de l'objet
Lec_num_genisse
        Lec_num_genisse.run() #ne termine que lorsqu'une boucle est lue
        num_genisse=Lec_num_genisse.num_gen #récupération de l'attribut
num_gen de Lec_num_genisse

        sequence(num_genisse)


if __name__=='__main__':
    try:
        main_fonction()
    except:
        main_fonction()
```

6.3 ANNEXE 3 : CODE D'EXTRACTION DES VALEURS DU ROI

ROI_process.c

```
#include <stdio.h>
#include <stdlib.h>

#include "/home/pi/RaspberryPi_v3/lepton_sdk/FLIR_I2C.h"

#include "/home/pi/RaspberryPi_v3/lepton_sdk/LEPTON_OEM.h"
#include "/home/pi/RaspberryPi_v3/lepton_sdk/LEPTON_SDK.h"
#include "/home/pi/RaspberryPi_v3/lepton_sdk/LEPTON_SYS.h"
#include "/home/pi/RaspberryPi_v3/lepton_sdk/LEPTON_Types.h"

#include "/home/pi/RaspberryPi_v3/lepton_sdk/LEPTON_RAD.h"


LEP_CAMERA_PORT_DESC_T lepton_port;


void get_roi()
// récupérer les données relatives au ROI et les enregistrées dans un
fichier txt
{
    LEP_RESULT result;
    LEP_RAD_SPOTMETER_OBJ_KELVIN_T roi_values;

    result = LEP_GetRadSpotmeterObjInKelvinX100(&lepton_port, &roi_values)
;
    printf(" LEP_GetRadSpotmeterObjInKelvinX100 = %d result =
%d.\n",roi_values.radSpotmeterMinValue , result);

}
int main(int argc, char **argv)
{
    LEP_OpenPort(1, LEP_CCI_TWI, 400, &lepton_port);
    get_roi ();
    return 0;
}
```

6.4 ANNEXE 4 : CODE D'ACQUISITION

lepton_data_collector.c

```
/*
 *  V4L2 video capture
 *
 *  This program can be used and distributed without restrictions.
 *
 *      This program is provided with the V4L2 API
 * see https://linuxtv.org/docs.php for more information
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include <getopt.h>             /* getopt_long() */

#include <fcntl.h>              /* low-level i/o */
#include <unistd.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

#include <linux/videodev2.h>

#include "lepton_vospi_funcs.h"

#define CLEAR(x) memset(&(x), 0, sizeof(x))

enum io_method {
        IO_METHOD_READ,
        IO_METHOD_MMAP,
        IO_METHOD_USERPTR,
};

struct buffer {
        void   *start;
        size_t  length;
};

static lepton_version   lep_version = LEPTON_VERSION_2X;
static char            *dev_name;
static enum io_method   io = IO_METHOD_MMAP;
static int              fd = -1;
struct buffer          *buffers;
static unsigned int     n_buffers;
static int              out_buf;
static char            *out_file_prefix = NULL;
static int              force_format;
static int              frame_count = 70;
static telemetry_location telemetry_loc = TELEMETRY_OFF;
```

```
static lepton_vospi_info lep_info;
static int              subframe_number = 0;
static int              frame_number = 0;
unsigned short         *pixel_data = NULL;

static void errno_exit(const char *s)
{
        fprintf(stderr, "%s error %d, %s\n", s, errno, strerror(errno));
        exit(EXIT_FAILURE);
}

static int xioctl(int fh, int request, void *arg)
{
        int r;

        do {
                r = ioctl(fh, request, arg);
        } while (-1 == r && EINTR == errno);

        return r;
}

static void process_image(const void *p, int size)
{
        if (out_buf)
        {
                FILE *out_file = NULL;
                char out_path[256];
                int done = 0;
                int lc_errs = 0;

                if (is_subframe_index_valid(&lep_info, (unsigned short
*)p)) {
                        lc_errs = extract_pixel_data(&lep_info, (unsigned
short *)p, pixel_data, &done);
                        if (done == 1) {
                                /* Either a Lepton 2.X frame was received,
or
                                 * the last subframe of a Lepton 3.X frame
was
                                 * received.
                                 */
                                snprintf(out_path, sizeof(out_path),
"%s%06d.gray", out_file_prefix,
                                        frame_number);
                                out_file = fopen(out_path, "wb");
                                if (!out_file) {
                                        /* indicate failure to store image
                                         * frame */
                                        fflush(stderr);
                                        fprintf(stderr, "G");
                                }
                                fwrite(pixel_data, sizeof(unsigned short),

lep_info.image_params.pixel_width*lep_info.image_params.pixel_height,
                                        out_file);
                                fclose(out_file);
                                frame_number++;
                        }
                }
```

```
                else {
                        /* indicate a throw-away frame, trying to sync on
                         * correct Lepton 3.x subframe.
                         */
                        fflush(stderr);
                        fprintf(stderr, "-");
                }
                if (lc_errs == 0) {
                        /* indicate raw frame stored successfully */
                        fflush(stderr);
                        fprintf(stderr, ".");
                }
                else {
                        /* indicate bad line counter failure in raw frame
*/
                        fflush(stderr);
                        fprintf(stderr, "%d", lc_errs);
                }
        }
        else {
                fflush(stderr);
                fprintf(stderr, ".");
                fflush(stdout);
        }
        subframe_number++;
}

static int read_frame(void)
{
        struct v4l2_buffer buf;
        unsigned int i;

        switch (io) {
        case IO_METHOD_READ:
                if (-1 == read(fd, buffers[0].start, buffers[0].length)) {
                        switch (errno) {
                        case EAGAIN:
                                return 0;

                        case EIO:
                                /* Could ignore EIO, see spec. */

                                /* fall through */

                        default:
                                errno_exit("read");
                        }
                }

                process_image(buffers[0].start, buffers[0].length);
                break;

        case IO_METHOD_MMAP:
                CLEAR(buf);

                buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                buf.memory = V4L2_MEMORY_MMAP;

                if (-1 == xioctl(fd, VIDIOC_DQBUF, &buf)) {
                        switch (errno) {
```

```
                        case EAGAIN:
                                return 0;

                        case EIO:
                                /* Could ignore EIO, see spec. */

                                /* fall through */

                        default:
                                errno_exit("VIDIOC_DQBUF");
                        }
                }
                if (buf.flags & V4L2_BUF_FLAG_ERROR) {
                        /* RLC - in case of data corruption, skip this
buffer
                        * and move on to the next.
                        */
                        fflush(stderr);
                        fprintf(stderr, "!");
                        fflush(stdout);
                        return 0;
                }

                assert(buf.index < n_buffers);

                process_image(buffers[buf.index].start, buf.bytesused);

                if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
                        errno_exit("VIDIOC_QBUF");
                break;

        case IO_METHOD_USERPTR:
                CLEAR(buf);

                buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                buf.memory = V4L2_MEMORY_USERPTR;

                if (-1 == xioctl(fd, VIDIOC_DQBUF, &buf)) {
                        switch (errno) {
                        case EAGAIN:
                                return 0;

                        case EIO:
                                /* Could ignore EIO, see spec. */

                                /* fall through */

                        default:
                                errno_exit("VIDIOC_DQBUF");
                        }
                }

                for (i = 0; i < n_buffers; ++i)
                        if (buf.m.userptr == (unsigned
long)buffers[i].start
                            && buf.length == buffers[i].length)
                                break;

                assert(i < n_buffers);
```

```
                process_image((void *)buf.m.userptr, buf.bytesused);

                if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
                        errno_exit("VIDIOC_QBUF");
                break;
        }

        return 1;
}

static void mainloop(void)
{
        unsigned int count;

        count = frame_count;

        while (frame_number < frame_count) {
                for (;;) {
                        fd_set fds;
                        struct timeval tv;
                        int r;

                        FD_ZERO(&fds);
                        FD_SET(fd, &fds);

                        /* Timeout. */
                        tv.tv_sec = 4; // réglage du timeout(initialement
=2)
                        tv.tv_usec = 0;

                        r = select(fd + 1, &fds, NULL, NULL, &tv);

                        if (-1 == r) {
                                if (EINTR == errno)
                                        continue;
                                errno_exit("select");
                        }

                        if (0 == r) {
                                fprintf(stderr, "select timeout\n");
                                exit(EXIT_FAILURE);
                        }

                        if (read_frame())
                                break;
                        /* EAGAIN - continue select loop. */
                }
        }
}

static void stop_capturing(void)
{
        enum v4l2_buf_type type;

        switch (io) {
        case IO_METHOD_READ:
                /* Nothing to do. */
                break;

        case IO_METHOD_MMAP:
```

```
        case IO_METHOD_USERPTR:
                type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                if (-1 == xioctl(fd, VIDIOC_STREAMOFF, &type))
                        errno_exit("VIDIOC_STREAMOFF");
                break;
        }
}

static void start_capturing(void)
{
        unsigned int i;
        enum v4l2_buf_type type;

        switch (io) {
        case IO_METHOD_READ:
                /* Nothing to do. */
                break;

        case IO_METHOD_MMAP:
                for (i = 0; i < n_buffers; ++i) {
                        struct v4l2_buffer buf;

                        CLEAR(buf);
                        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                        buf.memory = V4L2_MEMORY_MMAP;
                        buf.index = i;

                        if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
                                errno_exit("VIDIOC_QBUF");
                }
                type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                if (-1 == xioctl(fd, VIDIOC_STREAMON, &type))
                        errno_exit("VIDIOC_STREAMON");
                break;

        case IO_METHOD_USERPTR:
                for (i = 0; i < n_buffers; ++i) {
                        struct v4l2_buffer buf;

                        CLEAR(buf);
                        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                        buf.memory = V4L2_MEMORY_USERPTR;
                        buf.index = i;
                        buf.m.userptr = (unsigned long)buffers[i].start;
                        buf.length = buffers[i].length;

                        if (-1 == xioctl(fd, VIDIOC_QBUF, &buf))
                                errno_exit("VIDIOC_QBUF");
                }
                type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                if (-1 == xioctl(fd, VIDIOC_STREAMON, &type))
                        errno_exit("VIDIOC_STREAMON");
                break;
        }
}

static void uninit_device(void)
{
        unsigned int i;
```

```
        switch (io) {
        case IO_METHOD_READ:
                free(buffers[0].start);
                break;

        case IO_METHOD_MMAP:
                for (i = 0; i < n_buffers; ++i)
                        if (-1 == munmap(buffers[i].start,
buffers[i].length))
                                errno_exit("munmap");
                break;

        case IO_METHOD_USERPTR:
                for (i = 0; i < n_buffers; ++i)
                        free(buffers[i].start);
                break;
        }

        free(buffers);
}

static void init_read(unsigned int buffer_size)
{
        buffers = calloc(1, sizeof(*buffers));

        if (!buffers) {
                fprintf(stderr, "Out of memory\n");
                exit(EXIT_FAILURE);
        }

        buffers[0].length = buffer_size;
        buffers[0].start = malloc(buffer_size);

        if (!buffers[0].start) {
                fprintf(stderr, "Out of memory\n");
                exit(EXIT_FAILURE);
        }
}

static void init_mmap(void)
{
        struct v4l2_requestbuffers req;

        CLEAR(req);

        req.count = 4;
        req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        req.memory = V4L2_MEMORY_MMAP;

        if (-1 == xioctl(fd, VIDIOC_REQBUFS, &req)) {
                if (EINVAL == errno) {
                        fprintf(stderr, "%s does not support "
                                 "memory mapping\n", dev_name);
                        exit(EXIT_FAILURE);
                } else {
                        errno_exit("VIDIOC_REQBUFS");
                }
        }

        if (req.count < 2) {
```

```
                fprintf(stderr, "Insufficient buffer memory on %s\n",
                         dev_name);
                exit(EXIT_FAILURE);
        }

        buffers = calloc(req.count, sizeof(*buffers));

        if (!buffers) {
                fprintf(stderr, "Out of memory\n");
                exit(EXIT_FAILURE);
        }

        for (n_buffers = 0; n_buffers < req.count; ++n_buffers) {
                struct v4l2_buffer buf;

                CLEAR(buf);

                buf.type        = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                buf.memory      = V4L2_MEMORY_MMAP;
                buf.index       = n_buffers;

                if (-1 == xioctl(fd, VIDIOC_QUERYBUF, &buf))
                        errno_exit("VIDIOC_QUERYBUF");

                buffers[n_buffers].length = buf.length;
                buffers[n_buffers].start =
                        mmap(NULL /* start anywhere */,
                              buf.length,
                              PROT_READ | PROT_WRITE /* required */,
                              MAP_SHARED /* recommended */,
                              fd, buf.m.offset);

                if (MAP_FAILED == buffers[n_buffers].start)
                        errno_exit("mmap");
        }
}

static void init_userp(unsigned int buffer_size)
{
        struct v4l2_requestbuffers req;

        CLEAR(req);

        req.count  = 4;
        req.type   = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        req.memory = V4L2_MEMORY_USERPTR;

        if (-1 == xioctl(fd, VIDIOC_REQBUFS, &req)) {
                if (EINVAL == errno) {
                        fprintf(stderr, "%s does not support "
                                 "user pointer i/o\n", dev_name);
                        exit(EXIT_FAILURE);
                } else {
                        errno_exit("VIDIOC_REQBUFS");
                }
        }

        buffers = calloc(4, sizeof(*buffers));

        if (!buffers) {
```

```
                fprintf(stderr, "Out of memory\n");
                exit(EXIT_FAILURE);
        }

        for (n_buffers = 0; n_buffers < 4; ++n_buffers) {
                buffers[n_buffers].length = buffer_size;
                buffers[n_buffers].start = malloc(buffer_size);

                if (!buffers[n_buffers].start) {
                        fprintf(stderr, "Out of memory\n");
                        exit(EXIT_FAILURE);
                }
        }
}

static void init_device(void)
{
        struct v4l2_capability cap;
        struct v4l2_cropcap cropcap;
        struct v4l2_crop crop;
        struct v4l2_format fmt;
        unsigned int min;

        if (-1 == xioctl(fd, VIDIOC_QUERYCAP, &cap)) {
                if (EINVAL == errno) {
                        fprintf(stderr, "%s is no V4L2 device\n",
                                dev_name);
                        exit(EXIT_FAILURE);
                } else {
                        errno_exit("VIDIOC_QUERYCAP");
                }
        }

        if (!(cap.capabilities & V4L2_CAP_VIDEO_CAPTURE)) {
                fprintf(stderr, "%s is no video capture device\n",
                         dev_name);
                exit(EXIT_FAILURE);
        }

        switch (io) {
        case IO_METHOD_READ:
                if (!(cap.capabilities & V4L2_CAP_READWRITE)) {
                        fprintf(stderr, "%s does not support read i/o\n",
                                dev_name);
                        exit(EXIT_FAILURE);
                }
                break;

        case IO_METHOD_MMAP:
        case IO_METHOD_USERPTR:
                if (!(cap.capabilities & V4L2_CAP_STREAMING)) {
                        fprintf(stderr, "%s does not support streaming
i/o\n",
                                dev_name);
                        exit(EXIT_FAILURE);
                }
                break;
        }
```

```
        /* Select video input, video standard and tune here. */


        CLEAR(cropcap);

        cropcap.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

        if (0 == xioctl(fd, VIDIOC_CROPCAP, &cropcap)) {
                crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
                crop.c = cropcap.defrect; /* reset to default */

                if (-1 == xioctl(fd, VIDIOC_S_CROP, &crop)) {
                        switch (errno) {
                        case EINVAL:
                                /* Cropping not supported. */
                                break;
                        default:
                                /* Errors ignored. */
                                break;
                        }
                }
        } else {
                /* Errors ignored. */
        }


        CLEAR(fmt);

        fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        if (force_format) {
                fmt.fmt.pix.width       = 640;
                fmt.fmt.pix.height      = 480;
                fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
                fmt.fmt.pix.field       = V4L2_FIELD_INTERLACED;

                if (-1 == xioctl(fd, VIDIOC_S_FMT, &fmt))
                        errno_exit("VIDIOC_S_FMT");

                /* Note VIDIOC_S_FMT may change width and height. */
        } else {
                /* Preserve original settings as set by v4l2-ctl for
example */
                if (-1 == xioctl(fd, VIDIOC_G_FMT, &fmt))
                        errno_exit("VIDIOC_G_FMT");
        }

        /* Buggy driver paranoia. */
        min = fmt.fmt.pix.width * 2;
        if (fmt.fmt.pix.bytesperline < min)
                fmt.fmt.pix.bytesperline = min;
        min = fmt.fmt.pix.bytesperline * fmt.fmt.pix.height;
        if (fmt.fmt.pix.sizeimage < min)
                fmt.fmt.pix.sizeimage = min;

        switch (io) {
        case IO_METHOD_READ:
                init_read(fmt.fmt.pix.sizeimage);
                break;

        case IO_METHOD_MMAP:
```

```
                init_mmap();
                break;

        case IO_METHOD_USERPTR:
                init_userp(fmt.fmt.pix.sizeimage);
                break;
        }
}

static void close_device(void)
{
        if (-1 == close(fd))
                errno_exit("close");

        fd = -1;
}

static void open_device(void)
{
        struct stat st;

        if (-1 == stat(dev_name, &st)) {
                fprintf(stderr, "Cannot identify '%s': %d, %s\n",
                         dev_name, errno, strerror(errno));
                exit(EXIT_FAILURE);
        }

        if (!S_ISCHR(st.st_mode)) {
                fprintf(stderr, "%s is no device\n", dev_name);
                exit(EXIT_FAILURE);
        }

        fd = open(dev_name, O_RDWR /* required */ | O_NONBLOCK, 0);

        if (-1 == fd) {
                fprintf(stderr, "Cannot open '%s': %d, %s\n",
                         dev_name, errno, strerror(errno));
                exit(EXIT_FAILURE);
        }
}

static void usage(FILE *fp, int argc, char **argv)
{
        fprintf(fp,
                 "Usage: %s [options]\n\n"
                 "Version 1.3\n"
                 "Options:\n"
                 "-2 | --lepton2       Lepton 2.X selected\n"
                 "-3 | --lepton3       Lepton 3.X selected\n"
                 "-d | --device name   Video device name [%s]\n"
                 "-h | --help          Print this message\n"
                 "-m | --mmap          Use memory mapped buffers
[default]\n"
                 "-r | --read          Use read() calls\n"
                 "-u | --userp         Use application allocated buffers\n"
                 "-o | --output prefix Outputs frames to <filename
prefix><frame_no> files\n"
                 "-f | --format        Force format to 640x480 YUYV\n"
                 "-c | --count         Number of frames to grab [%i]\n"
```

```
                "-t | --telemetry     Telemetry location: one of 'off',
'start', or 'end'\n"
                "",
                argv[0], dev_name, frame_count);
}

static const char short_options[] = "23d:hmruo:fc:t:";

static const struct option
long_options[] = {
        { "lepton2", no_argument,       NULL, '2' },
        { "lepton3", no_argument,       NULL, '3' },
        { "device", required_argument, NULL, 'd' },
        { "help",   no_argument,       NULL, 'h' },
        { "mmap",   no_argument,       NULL, 'm' },
        { "read",   no_argument,       NULL, 'r' },
        { "userp",  no_argument,       NULL, 'u' },
        { "output", required_argument, NULL, 'o' },
        { "format", no_argument,       NULL, 'f' },
        { "count",  required_argument, NULL, 'c' },
        { "telemetry", required_argument, NULL, 't' },
        { 0, 0, 0, 0 }
};

int main(int argc, char **argv)
{
        int lepton_version_arg_found = 0;

        dev_name = "/dev/video0";

        for (;;) {
                FILE *test_f;
                int idx;
                int c;

                c = getopt_long(argc, argv,
                                short_options, long_options, &idx);

                if (-1 == c)
                        break;

                switch (c) {
                case 0: /* getopt_long() flag */
                        break;

                case '2':
                        lepton_version_arg_found++;
                        lep_version = LEPTON_VERSION_2X;
                        break;

                case '3':
                        lepton_version_arg_found++;
                        lep_version = LEPTON_VERSION_3X;
                        break;

                case 'd':
                        dev_name = optarg;
                        break;

                case 'h':
```

```
                        usage(stdout, argc, argv);
                        exit(EXIT_SUCCESS);

                case 'm':
                        io = IO_METHOD_MMAP;
                        break;

                case 'r':
                        io = IO_METHOD_READ;
                        break;

                case 'u':
                        io = IO_METHOD_USERPTR;
                        break;

                case 'o':
                        out_buf++;
                        out_file_prefix = optarg;
                        /* make sure the prefix itself is writable */
                        test_f = fopen(out_file_prefix, "wb");
                        if (!test_f) {
                                printf("Cannot open file name prefix '%s'
for writing.\n", optarg);
                                exit(1);
                        }
                        fclose(test_f);
                        /* delete the test file */
                        unlink(out_file_prefix);
                        break;

                case 'f':
                        force_format++;
                        break;

                case 'c':
                        errno = 0;
                        frame_count = strtol(optarg, NULL, 0);
                        if (errno)
                                errno_exit(optarg);
                        break;

                case 't':
                        if (strncmp("start", optarg, 5) == 0) {
                                telemetry_loc = TELEMETRY_AT_START;
                        }
                        else if (strncmp("end", optarg, 3) == 0) {
                                telemetry_loc = TELEMETRY_AT_END;
                        }
                        else if (strncmp("off", optarg, 3) == 0) {
                                telemetry_loc = TELEMETRY_OFF;
                        }
                        else {
                                printf("Unknown telemetry location '%s'\n",
optarg);
                                exit(1);
                        }
                        break;

                default:
                        usage(stderr, argc, argv);
```

```
                        exit(EXIT_FAILURE);
                }
        }

        if (lepton_version_arg_found > 1) {
                printf("Warning: Multiple lepton version command-line args
found. The last setting will be used.\n");
        }
        printf("Collecting frames from Lepton %d.X, ", (int)lep_version);
        if (telemetry_loc == TELEMETRY_OFF) {
                printf("telemetry off\n");
        }
        else if (telemetry_loc == TELEMETRY_AT_END) {
                printf("telemetry at end of subframe data\n");
        }
        else if (telemetry_loc == TELEMETRY_AT_START) {
                printf("telemetry at start of subframe data\n");
        }
        init_lepton_info(&lep_info, lep_version, telemetry_loc);
        pixel_data = (unsigned short
*)calloc(lep_info.image_params.pixel_width*lep_info.image_params.pixel_heig
ht,
                sizeof(unsigned short));
        if (!pixel_data) {
                errno_exit("Cannot allocate pixel buffer");
        }
        open_device();
        init_device();
        start_capturing();
        mainloop();
        stop_capturing();
        uninit_device();
        close_device();
        free(pixel_data);
        fprintf(stderr, "\n");
        return 0;
}
```

6.5 ANNEXE 5 : CARACTÉRISTIQUES TECHNIQUES FLIR A320

Imaging and Optical Data	
Field of View (FOV) / Minimum Focus Distance	25 x 18.8° / 0.4m
Focal Length	18mm
Spatial Resolution (IFOV)	1.36mrad
Lens Identification	Automatic
F-Number	1.3
Thermal Sensitivity/NETD	50mK @ +30°C
Image Frequency	30Hz
Focus	Automatic or manual (built-in motor)
Digital Zoom	1 - 8x continuous, interpolating zooming on images
Detector Data	
Detector Type	Focal plane array (FPA), uncooled microbolometer
Spectral Range	7.5 to 13µm
IR Resolution	320 x 240 pixels
Detector Pitch	25µm
Detector Time Constant	Typical 12ms
Measurement	
Measurement Functions	4 spotmeters, 4 areas (Box, max/min/average position), Isotherm (above, below, interval), Reference temperature, Temperature difference (between measurement functions, Reference temperature), measurement mask filter
Schedule Response	File sending (ftp), email (SMTP)
Measurement Corrections	Global and individual object parameters
Alarm	
Alarm Functions	6 automatic alarms on any selected measurement function, digital in, camera temperature, timer
Response	Digital out, log, store image, file sending (ftp), email (SMTP), notification

Storage of Images	
Image Storage Type	Built-in memory for image storage
File Formats	Standard JPEG, 16-bit measurement data included
Compatible with FLIR Software	ThermaCAM Researcher 2.9 TheraCAM Reporter 8 ThermaCAM QuickReport
Ethernet	
Ethernet Control	Control, result and image
Ethernet Type	100Mbps
Ethernet Standard	IEE 802.3
Ethernet Connector Type	RJ-45
Ethernet Communication	TCP/IP socket-based FLIR proprietary
Ethernet Video Streaming	MPEG-4, ISO/IEC 14496-1, MPEG-4 ASP@L5
Ethernet Power	Power over ethernet, PoE IEEE 802.3af class 0
Ethernet Image Streaming	16-bit 320 x 240 pixels: - Linear signal - Linear temperature - Radiometric
Ethernet Protocols	TCP, UDP, SNTP, RTSP, RTP, HTTP, ICMP, IGMP, ftp, SMTP, SMB (CIFS), DHCP, MDNS (Bonjour), uPnP
Digital Input/Output	
Digital Input Purpose	Image tag (start/stop/general), Input ext. device (programatically read)
Digital Input	2 opto-isolated, 10-30V DC
Digital Output Purpose	As function of ALARM, Output to ext. device (programatically set)
Digital Output	2 opto-isolated, 10-30 V DC, max 100mA
Digital I/O Isolation Voltage	500V RMS
Digital I/O Supply Voltage	12/24V DC, max 200mA
Digital I/O Connector Type	6-pole jackable screw terminal

Composite Video	
Video	Composite video output, PAL and NTSC compatible
Video Connector Type	Standard BNC connector
Power System	
External Power Operation	12/24V DC, 24W absolute max
External Power Connector Type	2-pole jackable screw terminal
Voltage	Allowed range 10-30V DC
Environmental Data	
Operating Temperature Range	-15 to 50°C
Storage Temperature Range	-40 to 70°C
Humidity (Operating and Storage)	IEC 60068-2-30/24 h 95% relative humidity +25 to 40°C
EMC	IEC 61000-6-2: 2001 (Immunity), EN 61000-6-3: 2001 (Emission), FCC 47 CFR Part 15 Class B (Emission)
Encapsulation	IP40 (IEC 60529)
Bump	25g (IEC 60068-2-29)
Vibration	2g (IEC 60068-2-6)
Physical Data	
Weight	0.7kg
Camera Size (L x W x H)	170 x 70 x 70mm
Tripod Mounting	UNC ¼"-20 (on three sides)
Base Mounting	2 x M4 thread mounting holes (on three sides)
Housing Material	Aluminium