Dragon Timelapse

Project update: 2-Nov-2015

Jeremy Fernsler

Project Description:

Create a dynamic time-lapse capture system that automatically adapts to the time of a running capture and uploads a completed file to a repository and runs on a Raspberry Pi.

PDF of this document located here.

Program Rundown:

Here is a list of current programs and their languages and dependencies. Full descriptions can be found further down:

- **timelapse.py** - *python*. Main controlling program
    - requires the picamera python package
- **ffmpegCmd** - *bash*. Handles compression in a subshell

    - requires FFMPEG to be installed
- **uploadCleanup.py** - *python*. Handles the SFTP upload, email status updates and directory cleanup.
    - requires pysftp python package
    - requires crypto python package
- **gpio.py** - *python*. Handles general purpose i/o tasks dealing with LEDs
- **capture.php** - *php*. Simple script that can alter a server side file
- **cam.config** - colon separated config file for static camera settings
- **server.config** - colon separated config file for user, pass and server settings for sftp and email

Code can be found at:
https://github.com/jfernsler/dragon-timelapse

Project Goals & Status:

- Capture images with overlaid timecode:
    - **This is working**.
    - The camera uses a mix of presets loaded from an external file and a sequence of settling the shutter and gains to reduce flickering in the time-lapse.
    - Timecode overlays happen at the moment of capture.
- Dynamic Capture
    - **This is working**.

    - Once a capture count of 1200 is reached, half of the files are deleted and the capture delay period is doubled. This ensures a video no longer than 40 seconds at 30 frames per second.
- Assembly of images into a quicktime
    - **This is working - with a caveat**
    - Originally this goal was to include the use of h.264 compression. However, while there is much information regarding the on board hardware acceleration for h.264, there is not a stable implementation using ffmpeg (the program for the compression).
    - h.264 is ideally a multipass operation and the ffmpeg implementation does not actually leverage this, which means the benefits of the compression format aren't fully realized.
    - Finally, h.264 is heavily asymmetrical - it's compression stage requires a significant amount of processing to complete and without any acceleration, the time to finalize a compression is unwieldy.
    - The compromise is to use MotionJPG as a compression scheme at full quality level. This gives a nearly visually lossless compression format which is highly compatible. It's lightweight enough to implement on the pi and the high quality of output is great.
- FTP to a remote location
    - **This is working**.
    - Instead of FTP, I have implemented SFTP.
    - This may have to be altered as we have removed FTP access to

our shared drive in and are replacing it with webDAV through OWNCloud.

- Clean up
    - **This is working**.
- Remote Start / Stop
    - **This is working**.
    - This aspect was a decent challenge as the remote server would occasionally revoke the connection. After various alterations, each one extending the amount of run time, it has now run for multiple days without issue.
    - The web connection is executed in a thread to prevent the program from hanging if it waits for a response.
    - The thread catches exceptions and then alters the check delay to allow the connection to fully reset. This has enabled very stable communications.
    - Right now it's very bare bones with a simple start / stop. The server side of things happens with a PHP script simply modifying a text file
    - I am working with someone in Interactive Digital Media to code up a more interesting page for their students to use.
- Launch on startup.
    - **Not complete**.
    - This will be the last item to happen after things are running smoothly
- Blinky Lights

    - **This is working**
    - Each state in the process is represented by a light:
        - Ready
        - Capturing
        - Compressing
    - Each of these can be addressed individually as well as queried for state
- Heartbeat
    - **This is working**
    - This one I just added because I've been thinking about it quite a bit given some of the reliance on outside sources (for uploading and activation). Initially I was thinking about maintaining a web based status but I've since implemented an occasional cycling of the lights as long as the system is still making URL requests.
    - The gpio.py program has a mode called Heartbeat which captures the current state of the GPIO pins, then does a quick cycling animation and restores the state. This happens every 2 minutes.

Code Outline:

- **timelapse.py** : main controlling program
    - Functions:

- main()
    - checks input for an arg to define maximum number of images in a capture sequence.
    - initiates waitToBegin()
- checkStatus()
    - launches a url request in search of a start or stop toggle
    - alters the global STATUS var
    - if the request fails, it catches the exception and creates a longer wait state in order to reset the socket.
- waitToBegin()
    - sits in a loop
    - generates a thread for checkStatus() after a predefined interval
    - once STATUS toggles to a start state, initiates runTimeLapse()
- setNames()
    - creates unique name for movie and capture dir
    - currently uses epoch time, eventually hoping to parse name from the STATUS file
- runTimelapse()
    - check for existence of and then creates a dir to capture to.
    - calls initCamera()
    - logs start time
    - enters loop to capture images on predetermined interval until STATUS is toggled to stop state

            - calls takePhoto()
            - captures until the max count is reached and calls removeEvenPics()
            - doubles the capture delay after max count is reached
            - generates a thread for checkStatus() after a predefined interval
        - logs completion time
        - calls compressFiles()
        - calls waitToBegin()
    - initCamera()
        - reads external cam.config file which contains settings which can be pre-defined
        - opens camera and allows time for exposure and white balance to settle and then locks in those settings to greatly reduce flickering content.
    - takePhoto()
        - sets up annotation information
        - captures image w/ passed number
    - removeEvenPics()
        - combs through capture directory and deletes every other image
        - calls renumberPics()
    - renumberPics()
        - combs though capture directory and re-sequences image

        numbers
    - compressFiles()
        - takes logged start stop data and image directory and passes it to the ffmpegCmd shell script with a system call.
- **ffmpegCmd** : retains the FFMPEG compression command, generates a log file of the activity.
    - calls uploadCleanup.py
    - by calling uploadCleanup.py from here it allows timelapse.py to start a new capture if necessary.
- **uploadCleanup.py** : performs an SFTP upload of the final movie file, cleans up the capture directory, emails a notification that a new movie is resident on the server
    - Functions:
    - main()
        - sequentially calls getConfig(), doUpload(), sendEmail(), and cleanUp()
    - getConfig()
        - reads in external file with all servers, userIDs and passwords and generates a global dictionary file
    - doUpload()
        - is passed arguments for the location of the movie file and a simple log
        - opens an SFTP connection to the sever
        - navigates to the proper directory and puts the files
        - captures a dir listing for a log file

        - closes connection
    - sendEmail()
        - initiates connection to the gmail SMTP server
        - compiles a notification email containing upload time and movie name
        - sends email to recipient listed in the server config file.
    - cleanUp()
        - deletes all image files and movie file
        - removes directory
- **gpio.py** : performs all GPIO related tasks
    - Functions:
    - main()
        - parses args
        - calls changeMode()
    - changeMode()
        - Alters the state of the LEDs based on the passed var
        - Current options:
            - READY - calls cyclefun() then turns on ready LED (pin 11)
            - COMPRESS - turns on compressing LED (pin 13)
            - CAPTURE - turns on capture LED (pin 15)
            - CAPTURECOMPLETE - turns off capture LED
            - COMPRESSCOMPLETE - turns off compress LED
            - NOTREADY - turns off read LED
            - Q - prints the status of all LEDs

        - HEARTBEAT - calls cyclefun() and then restores LED states
    - cyclefun()
        - loops twice cycling each LED on and off with a slight delay

---

Dynamic Capture Process

Images are initially captured at a preset delay (0.5 seconds) until 1200 images are created.

1	2	3	4	5	6	7	8	9	10	…	1200

Now, every other one is deleted - making the delay between existing images delay*2 (1 second)

1	**2**	3	**4**	5	**6**	7	**8**	9	**10**	…	**1200**

There are now 600 images in the directory with x2 numbers.

1	3	5	7	9	11	13	15	17	19	…	1199

These are renumbered to maintain an unbroken sequence (keeping FFMPEG happy). Capturing now continues with number 600 and the new delay set to delay*2. Maintaining original images and proper continuity.

1	2	3	4	5	6	7	8	9	10	…	600	…

This process may repeat indefinitely.