WhizniumSBE and WhizniumDBE

MPSI Technologies Modular Vision Demonstrator

Thermal Imager Data Path

A code walkthrough

September 19, 2018

Alexander Wirthmüller
aw@mpsitech.com

MPSI Technologies GmbH

Overview

This code walkthrough offers a basic understanding of embedded data processing using a combined WhizniumDBE and WhizniumSBE project of moderate complexity as an example.

The presented use case follows the data path of thermal images from their origin, a FLIR Lepton3 thermal imager, via FPGA-based read-out and Embedded Linux executable to a client's web-browser, see Figure 1.

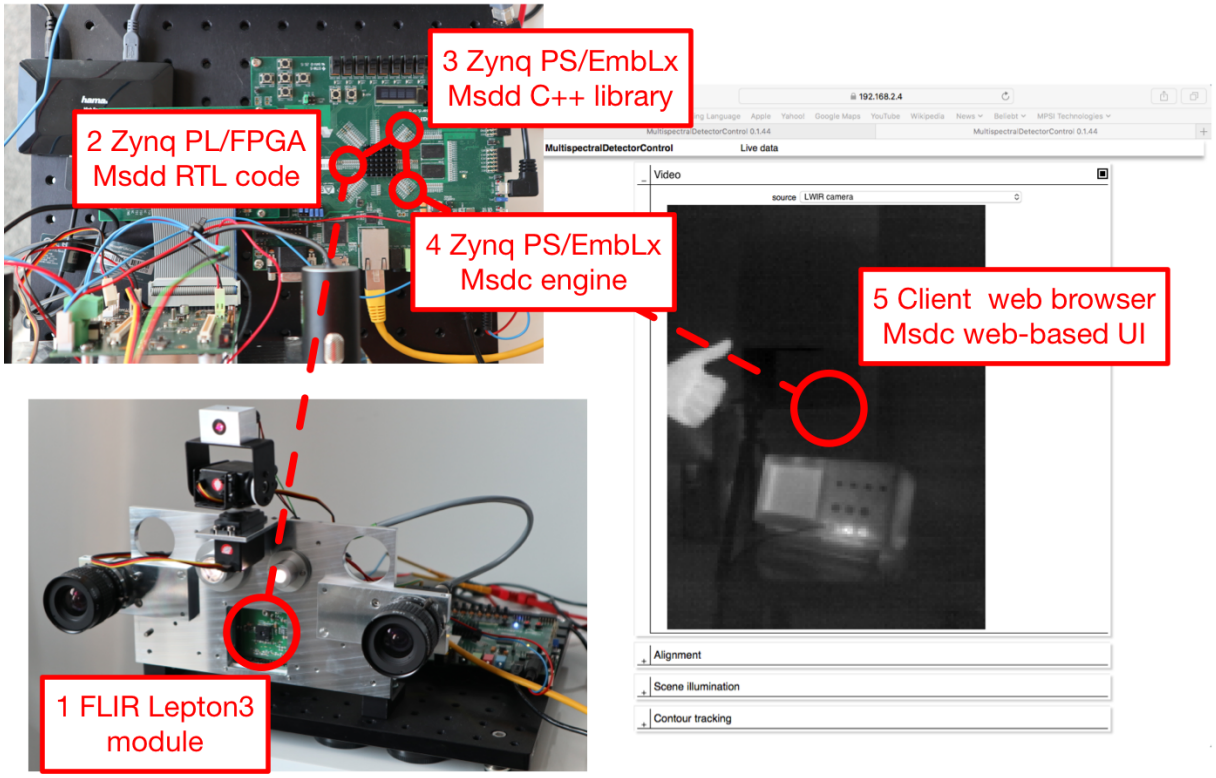

Figure 1: Five-step data path.

In order to follow this guide, it is suggested to download/clone the following repositories from the MPSI Technologies GitHub account:

- MultiSpectralDetectorDevice: modular vision demonstrator FPGA sub-system code https://github.com/mpsitech/MultiSpectralDetectorControl
- dbecore: WhizniumDBE core library[1] https://github.com/mpsitech/dbecore
- MultiSpectralDetectorControl: modular vision demonstrator Embedded Linux code https://github.com/mpsitech/MultiSpectralDetectorDevice
- sbecore: WhizniumSBE core library[1] https://github.com/mpsitech/sbecore

[1] dbecore and sbecore are suggested to follow the inline "excursions"

1 FLIR Lepton3 module

Within the 160x120 pixel FLIR Lepton3 thermal imager module, frame data is produced at a rate of 9fps. After module initialization via I2C, it is provided as a steady stream through a SPI interface.

This manufacturer datasheet provides details: https://www.flir.com/globalassets/imported-assets/document/flir-lepton-engineering-datasheet.pdf

Reset and master clock signals along with both serial interfaces are routed via custom hardware to a commercial Zynq evaluation board ("Zedboard"), where they end up at suitable FPGA I/O pads ; the hardware signals are detailed in Figure 2.

Lepton3 pin	**FPGA pad**	**purpose**
RESET_L	nirst (N17)	FPGA -> FLIR reset
MASTER_CLK	imclk (P21)	FPGA -> FLIR 25MHz clock

I2C 2-wire interface, "fast" mode 400kbps		
Lepton3 pin	**FPGA pad**	**purpose**
SCL	iscl (L17)	FPGA -> FLIR clock
SDA	isda (M17)	bi-directional push-pull data

SPI 3-wire interface, CPOL=1, CPHA=1, 12.5MHz		
Lepton3 pin	**FPGA pad**	**purpose**
SPI_CS_L	niss (N18)	FPGA -> FLIR chip select
SPI_CLK	isck (P20)	FPGA -> FLIR clock
SPI_MISO	irxd (T19)	FLIR -> FPGA data

Figure 2: Lepton3 hardware signals and their connections to the FPGA

2 Zynq PL / FPGA MultiSpectralDetectorDevice RTL code

Lepton3 configuration

`File: MultiSpectralDetectorDevice/msdd/zedb/Lwirif.vhd`

The main operation (`op`) FSM reacts to the command invocation `setRng(rng=tru8)` in its states `stateOpInit/Inv` using the host interface handshake ports `req/ackInvSetRng`.

A five second initialization time span, required by the Lepton3 module when coming out of reset with its master clock applied, is implemented using the unit's 10kHz signal `tkclk` (`stateOpStartA/B`).

Every five seconds, the Lepton3 read commands getSerno, getPartno, getAuxtemp, getFpatemp and getStats (sequence in `stateOpLoopCmd`) are triggered through the I2C

interface. Each read represents a 6-byte transfer in which one 16-bit data word is read from the module.

First the transfer length is transmitted to the Lepton3 module in `stateOpSetLenA/B`, followed by the command's base address in `stateOpSetCmdA/B`. If no module error is encountered, all bytes are burst-read in states `stateOpReadA/B`.

Each I2C transfer is activated using the combinatorial `stateOp`-dependent signal `reqI2c`.

`File: MultiSpectralDetectorDevice/msdd/zedb/I2c.vhd`

Transfer operation (`xfer`) FSM.

Frame acquisition

`File: MultiSpectralDetectorDevice/msdd/zedb/Lwiracq.vhd`

Main operation (`op`) FSM loops over segments and packets, filling ping-pong (A/B) buffer. {a/b}buf mutex management (`buf`) FSM takes care of A/B full/clear logic. {a/b}buf B/hostif-facing operation (`bufB`) FSM is used to output the correct data to the host interface.

`File: MultiSpectralDetectorDevice/msdd/zedb/Spimaster_v1_0.vhd`

Transfer operation (`xfer`) FSM. Strobe for every byte received.

`File: MultiSpectralDetectorDevice/msdd/zedb/Dpbram_v1_0_size38kB`

2x dual-port RAM connected to host interface.

Host interface

`File: MultiSpectralDetectorDevice/msdd/zedb/Zedb.vhd`

Command set constant definitions. Relevant for example:
`tixVControllerLwiracq/Lwirif`, `tixVLwiracqCommandSetRng`, `tixVLwiracqCommandGetInfo` (status polling), `tixWBuffer{Abuf/Bbuf}LwiracqToHostif`.

`File: MultiSpectralDetectorDevice/msdd/zedb/Hostif.vhd`

Main operation (`op`) FSM allows various types of transfers:

- `stateOpRxop` -> `stateOpRx` -> `stateOpTxack`:
  command with non-empty invocation parameters
- `stateOpRxop` -> `stateOpTx`:
  command with non-empty return parameters
- `stateOpRxop` -> `stateOpRxbuf` -> `stateOpTxack`:
  buffer transfer host to FPGA
- `stateOpRxop` -> `stateOpTxbuf`:
  buffer transfer FPGA to host

`File: MultiSpectralDetectorDevice/msdd/zedb/Crc8005_v1_0.vhd`

On-the-fly byte-wise CRC calculation in one clock cycle.

`File: MultiSpectralDetectorDevice/msdd/zedb/Axirx_v1_0.vhd, Axitx_v1_0.vhd`

AXI interconnect reacting to transfers initiated by the host (RX and TX) ; signals `enRx`, `rx`, `strbRx`, `enTx`, `strbTx` generated from original bus signals in wrapper module `Zedb_ip_v1_0_S00_AXI.vhd`.

3 Zynq PS / Embedded Linux MultiSpectralDetectorDevice C++ library

Status polling

`File: MultiSpectralDetectorDevice/msdd/devmsdd/UntMsddZedb/zedb/CtrMsddZedbLwiracq.cpp`

After starting the acquisition using void `setRng(const bool rng)`, the method `void getInfo(utinyint& tixVBufstate, uint& tkst, usmallint& min, usmallint& max)` - return parameters only, is invoked until the buffer state `tixVBufstate` (vector declaration in `CtrMsddZedbLwiracq.h`, `VecVBufstate`) reaches `abuf` or `bbuf`. At this time, `tkst/min/max` contain the frame metadata (10kHz clock time stamp, min/max values), and the frame data is available for buffer transfer.

`File: MultiSpectralDetectorDevice/msdd/devmsdd/UntMsddZedb/zedb/UntMsddZedb.cpp`

Byte-level command and buffer transfer communication is handled in `rx(unsigned char* buf, const size_t buflen)` and `tx(unsigned char* buf, const size_t buflen)`. The AXI interconnect is a Unix character device, allowing the use of standard `read()` / `write()` methods.

Excursion: Invocation parameters to byte sequence

`Files: dbecore/Cmd.cpp, dbecore/Par.cpp`

`Cmd::parsInvToBuf(unsigned char** buf, size_t& buflen)` and `Par::parsToBuf(map<string,Par>& pars, vector<string>& seqPars, unsigned char** buf, size_t& buflen)`.

Buffer transfer

`File: MultiSpectralDetectorDevice/msdd/devmsdd/UntMsddZedb/zedb/UntMsddZedb.cpp`

A frame comprises 38400 data bytes, which are followed by 2 CRC bytes in a single read transfer. The buffer transfer is initiated by a call to `void read{Abuf/Bbuf}FromLwiracq(const size_t reqlen, unsigned char*& data, size_t& datalen)` which in turn calls …

`File: MultiSpectralDetectorDevice/msdd/devmsdd/Msdd.cpp`

… `bool runBufxf(Bufxf* bufxf)` and `bool runBufxfFromBuf(Bufxf* bufxf)`. Memory allocation can be internal or external. Notably, on read operations from the FPGA subsystem, non-detection of erroneous all-zero transfers is avoided by inverting the CRC bytes.

4 Zynq PS / Embedded Linux MultiSpectralDetectorControl engine

LWIR acquisition thread and infinite loop

`File: MultiSpectralDetectorControl/msdc/msdccmbd/gbl/JobMsdcSrcMsdd.cpp`

The acquisition is started by a call to `bool startLwir(unsigned char* buf0, void (*callback)(void*), void* argCallback)`, to which a pointer to an (allocated) buffer for the first frame and a callback method + argument are passed.

WhizniumDBE's "easy" model does not support non-blocking operation when waiting for new frames, so that a separate thread with periodic polling is started.

The thread entry point is `void* runLwir(void* arg)` ; within its infinite loop, a buffer transfer is performed each time a new frame becomes available, after which the provided callback function is invoked.

Callback, acquisition and process stage

`File: MultiSpectralDetectorControl/msdc/msdccmbd/gbl/JobMsdcAcqLwir.cpp`

The callback function `void MsddCallback(void* arg)` is executed from within the thread mentioned before. It is thus crucial that it only performs the minimal actions required and then returns to the acquisition loop.

If an empty buffer is available for the next frame, the callback points the acquisition thread to it using `void JobMsdcSrcMsdd::setLwirBuf(unsigned char* buf)`, else a `NULL` pointer is passed, resulting in omitted frames. This condition is resolved once a buffer becomes available in `uint enterSgeAcq(DbsMsdc* dbsmsdc, const bool reenter)`.

Control over the newly arrived frame is passed on to one of the engine's job processor threads, by triggering an external call which is handled in `bool handleCallMsdcBufrdy(DbsMsdc* dbsmsdc, const ubigint jrefTrig, const boolvalInv)`. If the job's state machine is ready for a new frame, i.e. is in stage `WAITBUF`, the stage is changed to `ACQ`, else the frame is dropped.

In `uint enterSgeAcq(DbsMsdc* dbsmsdc, const bool reenter)`, an optional geometrical transform is applied to the frame, and also the raw data is auto-gain-corrected to spread over the entire 16-bit grayscale space.

Acquisition takes place in the master job, and triggering a `CallMsdcIbitRdy` notifies all slaves of the now pre-processed frame.

Both the master job an all slave jobs subsequently switch into the `PRC` stage, where each instance of `JobMsdcAcqLwir` can perform the additional geometrical transforms it needs.

Finally, also in `uint enterSgePrc(DbsMsdc* dbsmsdc, const bool reenter)`, a `CallMsdcImgRdy` is triggered which notifies the superior (UI) panel job of the processed frame.

Live data video panel

`File: MultiSpectralDetectorControl/msdc/msdccmbd/CrdMsdcLiv/PnlMsdcLivVideo.cpp`

In `bool handleCallMsdcImgrdy(DbsMsdc* dbsmsdc, const ubigint jrefTrig)`, the frame data is copied into a `DpchEngLive` dispatch (definition in `PnlMsdcLivVideo_blks.cpp`), a serializable (to XML) C++ object. As the following HTTPS transfer to the client is to be initiated by the server (at least virtually / emulated), the dispatch is passed on to a dispatch collector up the job hierarchy.

Dispatch collector for long polling

`File: MultiSpectralDetectorControl/msdc/msdccmbd/Msdccmbd.cpp`

HTTP/1.1 does not support communication triggered by the server for which reason WhizniumSBE applications use “long polling” (delayed answering of a client request to emulate server-triggered action). Per card or browser tab, in this case for `CrdMsdcLiv`, a dispatch collector accumulates pending dispatches while no client request is available.

In the engine’s exchange object, `void submitDpch(DpchEngMsdc* dpcheng)` handles the matching of dispatches to be transferred to available dispatch collectors. If a (now obsolete) `DpchEngLive` is alrady present, its content is overwritten/merged with the new LWIR frame.

If a HTTPS request is available, control will move over to the application server, else the dispatch will be retained.

HTTPS application server powered by libmicrohttpd

`File: MultiSpectralDetectorControl/msdc/msdccmbd/MsdccmbdAppsrv.cpp`

Dispatch collector (“notify”) HTTPS/GET requests are received at the URL `https://<ip>:<port>/notify/<scrJref>` where `<scrJref>` is the scrambled job reference, in this case of the relevant instance of `CrdMsdcLiv`.

In the application server’s libmicrohttpd callback function, `int MhdCallback(void* cls, MHD_Connection* connection, const char* url, const char* method, const char* version, const char* upload_data, size_t* upload_data_size, void** con_cls)` and then `void`

`writeDpchEng(XchgMsdccmbd* xchg, ReqMsdc* req)`, the dispatch is serialized into a XML string using its `void writeXML(const uint ixMsdcVLocale, xmlTextWriter* wr)` method.

Excursion: Base64 encoding for XML

`Files: sbecore/Xmlio.cpp`

Binary data (8-bit character space) is reduced to Base64 encoding (6-bit character space) in `void toBase64(const unsigned char* inbuf, unsigned int inbuflen, char** outbuf, unsigned int& outbuflen)` in order to be transmitted correctly. Also, machine type (e.g. Intel little-endian, ARM big-endian) independence is warranted by always using "network order" (big-endianness) for the transfer of multi-byte variables, this is implemented in `void writeBase64(xmlTextWriter* wr, const char* _buf, const unsigned int len, const unsigned int varlen)`.

5 Client web-browser MultiSpectralDetectorControl web-based UI

HTML5 canvas element

`File: MultiSpectralDetectorControl/msdc/webappmsdc/CrdMsdcLiv/PnlMsdcLivVideo_bcont.xml`

In terms of WhizniumSBE modeling, the area in which the LWIR image is to be displayed, is a custom control `CusImg` in row `<tr id="trImg">` of height 480 pixels. A HTML5 canvas element is inserted manually, its bitmap content is accessible through JavaScript code.

Long-polling the engine's application server

`File: MultiSpectralDetectorControl/msdc/webappmsdc/CrdMsdcLiv/CrdMsdcLiv.js`

The function `iterateReqit()` forms the counterpart to the engine's dispatch collector. The `responseXML` representation of the data received already allows to interpret the dispatch XML root tag. Each card knows its panel's scrambled job references, allowing to pass on a received dispatch accordingly (here to `PnlMsdcLivVideo`).

Image display

`File: MultiSpectralDetectorControl/msdc/webappmsdc/CrdMsdcLiv/PnlMsdcLivVideo.js`

The `DpchEngLive` arrives at `handleDpchEng(dom, dpch)` and then `handleDpchEngMsdcLivVideoLive(dom)`. Its non-binary content is inserted into the panel's DOM, while the grayscale bitmap is transformed back into binary form and subsequently stored in the variable `doc.gray`.

Along with some scaling, the function `refreshLive(mask)` is responsible for updating the canvas RGBX bitmap content.

Excursion: Base64 decoding

`File: MultiSpectralDetectorControl/msdc/webappmsdc/script/vecio.js`

Function `fromBase64(str)`. As JavaScript does not assume “network order” but rather the local machine’s endianness, also an optional re-ordering is implemented in all `parse*(str)` functions.