A. Overview

This is a brief description of STM32CubeMX project ‘mxusartusbcan’ running a DiscoveryF4 board.

The goal of ‘mxusartusbcan’ was for development, debugging, & testing of routines that interface the winch and related projects to the STM32CubeMX w FreeRTOS and determine if the STM32 HAL routines were usable without modifications.

The general conclusion is that the HAL routines w FreeRTOS are usable, but the overhead is substantially higher than with our bare-metal routines and use of interrupt levels in place of a RTOS.

B. Tasks

A brief description of the tasks, by task name, in ‘mxusartusbcan’.

1. CanTask01

Location: main Priority: osPriorityIdle (-3)

Generated by: Manual

- Task generated manually (rather than via STM32CubeMX)

- Endless loop pacing: osDelay of 5 seconds

- Lists stack high-water mark for all the FreeRTOS tasks

- Test sending serial output to usart6 using ‘yprintf’ and multiple buffers

- Test that floating point is being linked with linker and a test message with fp working

- Toggle Blue LED each loop.

2. DefaultTask

Location: main Priority: osPriorityNormal (0)

Generated by: STM32CubeMX

- Test usb-cdc (i.e. ttyACM0) sending “serial” data.

- Test sending short bursts of a fixed CAN message

- Toggle Green LED

Note: normally this task would be the assigned the lowest FreeRTOS task priority, but in this case CanTask01 ended being the lowest to make the stack-high-water-mark a background type of task.

3. Task02

Location: main Priority: osPriorityIdle (-3)

Generated by: STM32CubeMX

- Endless loop: Wait on notification from ADC or incoming serial line ready.

- Test ‘yscanf’ (a wrapper for vnscanf w semaphore and buffers) for incoming ASCII lines from usart6.

- Test ADC/DMA: threeADC channels plus internal temperature and voltage reference.

- Test using multiple notification bits (xTaskNotifyWait) with yscanf, ADC processing, and incoming serial “lines.”

4. Task03

Location: main Priority: osPriorityIdle (-3)

Generated by: STM32CubeMX

- Endless loop: Wait on received CAN messages placed in a queue by ‘iface.c’.

- Test receiving CAN messages from ‘can\_iface.c’ (interface to HAL CAN routine).

- Convert incoming CAN messages to ASCII/HEX format and send to PC via usart2 running at 2E6 baud (same format as our standard gateway).

- Set a one second tick flag from CAN bus logger/GPS message (0x00400000)

- Count incoming CAN messages and display count via usart6 each second, which provides additional testing of multiple tasks sending output to the SerialTaskSend.

5. CanTxTask

Location: Ourwares/CanTask Priority: 5

Generated by: Manual

Brief: a created queue receives CAN messages from other tasks and sends those CAN messages to ‘can\_driver\_put’ in the ‘can\_iface.c’ routine, which adds the message to a linked list, sorted in CAN priority, and aborts a pending lower priority CAN message if the newly added CAN message is of higher priority.

Orange LED toggling is used for debugging.

6. CanRxTask

Location: Ourwares/CanTask Priority: 3

Generated by: Manual

Brief: a created queue receives incoming CAN messages. The code to place selected CAN messages in mailboxes has not been added, so the RxTask is placed “suspended.” However, the queue is used by ‘can\_iface’ to send CAN messages which are retrieved by Task03.

7. SerialTaskReceive

Location: Ourwares/CanTask Priority: 0

Generated by: Manual

Brief: The Task handles polling and notifications from DMA interrupts, unloading incoming ASCII from the DMA buffer into a circular buffer of lines (i.e. a line is a serial input stream that ends with a CR or LF). For char-by-char mode the line chars are added to the line buffering in the interrupt service. For DMA the task waits for a osDelay tick of one, or a notification from the interrupt service by the DMA of the ½ and end of buffer interrupts. The task unloads the accumulated chars in to line buffers and notifies the originating task when a line completes.

Only one Task per uart, i.e. there is no provision for distributing completed lines to multiple Tasks.

Addition planned: creating a (binary) CAN message, rather than a “line” from input expected to be the ascii/hex format used in the PC<→gateway communication. Without this there would be a double passing of each incoming char, first to make a line, then second to convert to the CAN message.

8. SerialTaskSend

Location: Ourwares/CanTask Priority: 0

Generated by: Manual

Brief: During initialization a queue is created that receives buffer control blocks (bcb) from multiple tasks. Each uart is added to a linked list with an associated circular buffer of bcb pointers for that uart. Tasks originating with data to be sent create buffers w buffer control blocks, where the appropriate size of the byte buffer is specified.

A semaphore is created for the buffer and included in the bcb. The semaphore is used to block a task from reloading a buffer that has been queued to be sent, but sending has not completed.

During the endless loop in a task the buffer is loaded and a pointer to the buffer control block is placed on the queue for SerialTaskSend. SerialTaskSend removes the pointer from the queue and adds it to the circular buffer (of pointers to buffer control blocks) for the uart associated with that buffer. The serial output to that uart is started if it is not already busy.

Interrupt callbacks “gives” the originating the buffer the semaphore. If more bcb in the circular buffer remain, it starts the sending of next buffer.

This allows a task to load a buffer and continue processing while the buffer is queued for sending. Before loading the buffer the semaphore “Take” blocks the task until the previous operation with the buffer has completed.

If multiple output “lines” are to be sent in quick succession and the same buffer is used, the task will likely be in a blocked state waiting for the buffer sending to complete and the buffer released. Using multiple buffers allows the task to continue.

This works for char-by-char and dma (circular) modes for uart/usarts. The difference between the two modes is essentially transparent to the application task.

B. “Ourwares” routines

A directory “Ourwares” is at the project directory level along with “Middlewares” and “Drivers” created by STM32CubeMX. The objective was to include routines that are likely to be needed in any of the projects we might do using STM32CubeMX. (“Ourtasks” directory is for more project specific needs and a means to offload code from ‘main’ which is rather large as generated from STM32CubeMX.) The following is a brief description of the “Ourwares” tasks.

1. getserialbuf.c

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* struct SERIALSENDTASKBCB\* getserialbuf(\

UART\_HandleTypeDef\* phuart,\

uint16\_t maxsize,\

uint32\_t notebit,\

uint32\_t\* pnoteval);

\* @brief : Create a buffer control block (BCB) for serial sending

\* @param : phuart = usart handle (pointer)

\* @param : size = number of uint8\_t bytes for this buffer

\* @param : notebit = single bit used for notification of this buffer

\* @param : pnoteval = Pointer to Task word that receives notification

\* @return : pointer to BCB; NULL = failed

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

Example:

In ‘main’ and task:

void StartTask02(void const \* argument)

...

/\* Get buffer with buffer control block and semaphore for use with 'yprintf' \*/

struct SERIALSENDTASKBCB\* pbuf1 = getserialbuf(&huart6,128);

struct SERIALSENDTASKBCB\* pbuf3 = getserialbuf(&huart2,128);

...

/\* Convert binary CAN to ASCII/HEX format for PC \*/

xSemaphoreTake(pbuf3->semaphore, 5000);

gateway\_CANtoPC(&pbuf3, &ncan.can);

vSerialTaskSendQueueBuf(&pbuf3); // Place on queue

...

/\* Upon GPS CAN msg, 1/64 tick == 0, output received CAN msg count. \*/

yprintf(&pbuf1,"\n\r\t\t=====> R %i %i %i %i %i", rctr,\

(unsigned int)(debug1-debug1prev),\

(unsigned int)(debugctr-debugctr\_prev),\

(unsigned int)(debugTX1b - debugTX1b\_prev),\

(unsigned int)(debugTX1c - debugTX1c\_prev));

The calls to ‘getserialbuf’ creates byte buffers and buffer control blocks from the heap and returns a pointer to the buffer control block. These calls are made after a Task starts so that the Task Handle from the currently running task can be retrieved and included in the buffer control block.

For sending, a pointer to the buffer control block pointer (pointer-to-pointer) is used, e.g. in StartTask02 a pointer to pbuf3 ends up placed on the queue to SerialTaskSend--

yprintf(&pbuf1,"\n\r ...

2. yprintf.c

‘yprintf.’ and ‘yputs’ are two routines in ‘yprintf.c’

‘yprintf’ provides a means of making printf types of output from multiple tasks to multiple uarts.

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* int yprintf(struct SERIALSENDTASKCB\*\* ppbcb, const char \*fmt, ...);

\* @brief : 'printf' for uarts

\* @param : pbcb = pointer to pointer to stuct with uart pointers and buffer parameters

\* @param : format = usual printf format

\* @param : ... = usual printf arguments

\* @return : Number of chars "printed"

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

This is a wrapper for ‘vsnprintf’ which includes a semaphore for printf and a buffer semaphore for the checking if the buffer is busy.

The first argument is a pointer to the pointer that points to the buffer control block. The following arguments are the same as one would use for the standard printf.

The semaphore is used to block other tasks from interrupting execution of the vsnprintf. Multiple tasks can call yprintf, but will block until vsnprintf completes for the originating task.

‘vsnprintf’ limits the number of bytes stored in the buffer. This count is supplied by the buffer control block. If the number of bytes generated by the vsnprintf exceeds the buffer size the output line is truncated to the length of the buffer.

‘yputs’ provides copies a string to the bcb buffer and handles the semaphore for the buffer.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/\*\*\*

\* int yputs(struct SERIALSENDTASKYCB\* pbcb, char\* pchr);

\* @brief : Send zero terminated string to SerialTaskSend

\* @param : pbcb = pointer to pointer to stuct with uart pointers and buffer parameters

\* @return : Number of chars sent

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

‘yputs’ is included in ‘yprintf.c’ and has the same characteristics as ‘yprintf’ except it copies the input string to the output buffer instead of using printf.

3. can\_iface.c

This lifts some of the code from the bare-metal routine ‘can\_driver’. The key feature is the implementation of the linked list that is in sorted CAN priority order, and the aborting/replacing of a new, higher priority CAN message than one pending in the hardware register.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int can\_driver\_put(struct CAN\_CTLBLOCK\* pctl, struct CANRCVBUF \*pcan, u8 maxretryct, u8 bits);

/\* @brief : Get a free slot and add CAN msg

\* @param : pctl = pointer to control block for this CAN modules

\* @param : pcan = pointer to msg: id, dlc, data (common\_can.h)

\* @param : maxretryct = 0 = use TERRMAXCOUNT; not zero = use this value.

\* @param : bits = Use these bits to set some conditions (see .h file)

\* @return : 0 = OK;

\* : -1 = Buffer overrun (no free slots for the new msg)

\* : -2 = Bogus CAN id rejected

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

‘can\_iface.c’ is a CAN interface routine that goes between CanTxTask and the HAL stm32f4xx\_hal\_can.c routine.

In ‘main’, in the task ‘StartDefaultTask’ contains an example of sending CAN messages. A CAN message struct is placed on the CAN Tx queue, e.g.--

xQueueSendToBack(CanTxQHandle,&testtx,portMAX\_DELAY);

In ‘CanTask.c’ the function ‘StartCanTxTask’ removes CAN messages from the queue and sends them to ‘can\_driver\_put’ in ‘can\_iface.c’, which places the CAN message on a linked list in CAN priority sorted order. It also starts the CAN message sending if the HAL CAN routine is not busy. If the added CAN message is of higher priority than a pending CAN message in the hardware register it aborts the message in the hardware register. The abort interrupt loads the CAN message from the top of the linked list, which is the highest priority message. This prevents a low priority message waiting for other higher priority traffic on the CAN bus blocking the high priority message that was just added to the list.

To implement the CAN message priority scheme a single CAN TX mailbox is used. After a CAN message has been sent, to have the next CAN message loaded in time for the next CAN bus start-of-frame, requires servicing the interrupt in less than six CAN baud ticks, i.e. six 2 us ticks at 500KHz baud. Therefore, the CAN Tx interrupt should be high. However, since it also makes FreeRTOS API calls it cannot be higher than the max FreeRTOS level. Note that the hardware priority interrupt numbering is reverse of the FreeRTOS task priority numbering. E.g. if the max FreeRTOS level is 5, then the CAN interrupt setting could be 5, 6, … 15, with 5 the highest priority, and 15 the lowest.

‘iface.c’ has a #define that will enable outgoing CAN messages to be placed on the queue with incoming CAN messages. This might be needed if the outgoing CAN messages would be passed to the PC via a uart, along with incoming CAN messages. However if this is done, note that any messages sent by the PC would loopback to the PC.

#define CANMSGLOOPBACK

‘iface.c’ places incoming CAN messages on the receive queue. One (and only one) task removes the messages from the queue, e.g. in ‘main’ in the task ‘StartTask03’ the endless loops checks for incoming CAN messages--

Qret = xQueueReceive(CanRxQHandle,&ncan,portMAX\_DELAY);

In ‘StartTask03’, for test, the CAN messages are counted and converted to the ASCII/HEX format used in the PC. These are sent via usart2 at 2E6 baud. In the PC ‘socat’, ‘hub-server’ and ‘cangate’, ‘nc’, etc., can be used.

Note that in ‘main’ the STM32CubeMX generated CAN initialization fails if the CAN bus is not plugged in to the CAN module being initialized. This results in a call to--

void Error\_Handler(void)

To check if the CAN initialization was successful the Error\_Handler some user code stores the stack pointer. Before the endless loop started, a task printed these out, but that code is not in the present version of ‘mxusartusbcan’.

4. adctask.c

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* adctask\_init(ADC\_HandleTypeDef\* phadc,\

uint32\_t notebit1,\

uint32\_t notebit2,\

uint32\_t\* pnoteval,\

uint16\_t dmact);

\* @brief : Setup circular line buffers this uart

\* @param : phadc = pointer to ADC control block

\* @param : notebit1 = unique bit for notification @ 1/2 dma buffer

\* @param : notebit2 = unique bit for notification @ end dma buffer

\* @param : pnoteval = pointer to word receiving notification word from OS

\* @param : dmact = number of sequences in 1/2 of circular DMA buffer

\* @return :

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

A call to ‘adctask\_init’ starts the ADC running in a dma mode. Once started it runs continuously. STM32CubeMX setup determines which and how many ADCs are read. The current ‘mxusartusbcan’ has five ADCs, with the first three IN0, IN4, IN5, internal temperature, internal voltage reference.

The dma interrupts at the ½ buffer and end of buffer marks. Two notification bits are used; one for the ½ mark interrupt and the other for the end. This allows the task working with the readings to set a pointer to the dma buffer half that was just completed.

In ‘main’ in ‘StartTask02’ the ADC handling is demonstrated. A call to ‘adctask\_sum’ in ‘adctask.c’ is made to sum the adc readings in ½ of the dma buffer. These summed readings are further summed to reduce the rate, as well as average the readings. The final sums are converted to floats and scaled for output.

C. Notifications

Notifications from multiple sources are demonstrated in ‘StartTask02’. In this task the following notification bits are defined--

#define TSK02BIT02 (1 << 2) // Task notification bit for ADC dma 1st 1/2 (adctask.c)

#define TSK02BIT03 (1 << 3) // Task notification bit for ADC dma end (adctask.c)

The ADC sequence is fast with just five ADCs being read and operations sending serial output is relatively slow, so multiple buffers are used for the output. With one buffer multiple lines would stall the loop waiting for the buffer to become free and ADC notifications would be lost and/or the dma buffer not summed before being overwritten.

When the endless loop starts it waits for any notification to this task--

xTaskNotifyWait(noteused, 0, &noteval, portMAX\_DELAY);

When a notification(s) arrive and ‘noteval’ receives a copy from the os notification word holds with the bit(s) OR’ed in . These bits are tested and the appropriated action taken. When a bit from ‘noteval’ is tested and action taken, that bit is OR’d into ‘noteused’, i.e. the bit was “used.” When the loop ends and re-enters the TaskNotifyWait ‘noteused’ is provided as the first argument, which resets the bits in the os’s internal notification word. Additional notifications could have arrived between the last check of ‘noteval’ (the notification word copy received from the os and the entering of the TaskNotifyWait, so ‘noteused’ will only reset those notification bits that have been processed. If other notification bits are on TaskNotifyWait will immediately exit and those bits handled.

‘Task02’ in ‘mxusartusbcan’ uses multiple notifications. The ADC DMA makes notifications when the ½ buffer and end-of-buffer DMA interrupts are serviced. The SerialTaskReceive also makes a notification when an incoming line on usart6 has been completed. When ‘yprintf’ is called it waits if the buffer has not been released from the previous operation. If the time required to output a line exceeds the time between the notifications (e.g. ADC, and input lines) then notifications could be lost. Generally, the single line output is fast enough to not be a problem, but a sequence of multiple lines would likely require multiple buffers. Net, the mixing of serial output with other high rate processes in the same task needs consideration of the timing.

D. ‘mxusartusbcan’ initialization

The bulk of ‘main’ is initialization. The above mentioned tasks generated by STM32CubeMX place the tasks in ‘main’ with an empty “for ( ;; )” loop. In ‘mxusartusbcan’ the user task code is in ‘main’, but it also makes a large file even larger. Placing tasks code, and even the tasks in “Ourtask” with separate files for each task would help modularize the “main mess.”

E. Makefile modifications

When generating code in STM32CubeMX select ‘Makefile’ from the list. Once the project has been generated some additions and changes are needed. Once these changes are made, regeneration of the code by STM32CubeMX does not affect the changes. However, I did add “USER CODE” lines in Makefile, partly to make it easier to locate the additions.

1. Add files to be compiled

Add files to be compiled in the “C sources” section of ‘Makefile’ following the generated lines. Note that STM32CubeMX ends the line with ‘\’ to continue the next line, whereas our added files use the form C\_SOURCES += .

Drivers/STM32F4xx\_HAL\_Driver/Src/stm32f4xx\_hal\_adc.c

######################################

# source

######################################

# C sources

C\_SOURCES = \

Src/main.c \

...

Drivers/STM32F4xx\_HAL\_Driver/Src/stm32f4xx\_hal\_adc.c \

Drivers/STM32F4xx\_HAL\_Driver/Src/stm32f4xx\_hal\_adc\_ex.c

# /\* USER CODE BEGIN \*/

C\_SOURCES += Ourwares/SerialTaskSend.c

C\_SOURCES += Ourwares/cdc\_txbuff.c

C\_SOURCES += Ourwares/DTW\_counter.c

C\_SOURCES += Ourwares/CanTask.c

C\_SOURCES += Ourwares/can\_iface.c

C\_SOURCES += Ourwares/canfilter\_setup.c

C\_SOURCES += Ourwares/getserialbuf.c

C\_SOURCES += Ourwares/yprintf.c

C\_SOURCES += Ourwares/USB\_PC\_gateway.c

C\_SOURCES += Ourwares/PC\_gateway\_comm.c

C\_SOURCES += Ourwares/gateway\_comm.c

C\_SOURCES += Ourwares/gateway\_CANtoPC.c

C\_SOURCES += Ourwares/SerialTaskReceive.c

C\_SOURCES += Ourtasks/stackwatermark.c

C\_SOURCES += Ourtasks/yscanf.c

C\_SOURCES += Ourtasks/adctask.c

# /\* USER CODE END \*/

2. Includes

Add the directories of “Ourware” and “Ourtasks” to the includes section--

# C includes

C\_INCLUDES = \

-IInc \

…

-IMiddlewares/Third\_Party/FreeRTOS/Source/CMSIS\_RTOS \

-Idrivers/CMSIS/Include

# /\* USER CODE BEGIN \*/

C\_INCLUDES += -IOurwares

C\_INCLUDES += -IOurtasks

# /\* USER CODE END \*/

3. Linker flags

Changes are needed to the linker flags if floating point output (printf) and input (sscanf) is needed. These add substantially to the code size. For sscanf input, approximately 18.5K is added to the binary output, and less likely to be needed than printf fp output.

#######################################

# LDFLAGS

#######################################

# link script

…

LDFLAGS = $(MCU) -u \_printf\_float -T$(LDSCRIPT) $(LIBDIR) $(LIBS) -Wl,-Map=$(BUILD\_DIR)/$(TARGET).map,--cref -Wl,--gc-sections

To provide floating point output in printf add--

-u \_printf\_float

To provide floating point input for sscanf remove--

-specs=nano.specs

F. Build and Flashing

A script ‘mm’ is used to compile and flash. With the following changes, the compile and flash is executed by entering ‘./mm’ on the command line.

Change the name, e.g. ‘mxusartusbcan’ in the script below, to the name at the beginning of the ‘Makefile’ which is set when STM32CubeMX generates the code.

export FLOAT\_TYPE=hard

make

./script-all mxusartusbcan

‘script-all’ generates the .list and .srec files, and ‘script-all’ calls ‘script-flash’ which flashes the program using openocd and the .srec file. ‘Makefile’ generates a .bin file, which could be used with openocd, but all the prior scripts with the bare-metal work used the srec format, so as long as a .list file was generated, adding the .srec was a minor addition.

‘script-flash’ may need changing of the paths to match where openocd has been stored--

OPENOCD\_SCRIPTS=$OPENOCD\_BASE/openocd/scripts

OPENOCD\_FLASHER=$OPENOCD\_SCRIPTS/interface/stlink-v2.cfg

OPENOCD\_BOARD=$OPENOCD\_SCRIPTS/board/stm32f4discovery.cfg# Paths for openocd and automatic flashing

# The following two are 'export'ed in .arm\_tools via .profile

#OPENOCD\_BASE = /usr/share

#OPENOCD = /usr/bin/openocd

OPENOCD\_SCRIPTS=$OPENOCD\_BASE/openocd/scripts

OPENOCD\_FLASHER=$OPENOCD\_SCRIPTS/interface/stlink-v2.cfg

OPENOCD\_BOARD=$OPENOCD\_SCRIPTS/board/stm32f4discovery.cfg

G. Debugging

Placing the following script line in a directory in the PATH, e.g. ‘bin’, and giving it a name simplifies starting the stlink. Execute the following in a separate terminal window--

openocd -f interface/stlink-v2.cfg -f board/stm32f4discovery.cfg

With a usb cable plugged in to the DiscoveryF4 STLink usb port, execute the above script line. If the startup is successful it will list a number of lines. The last two lines, shown below, means success--

...

Info : Target voltage: 3.254183

Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints

In a different window, usually the one used for the compiling, start the debugging client--

telnet localhost 4444

Success is--

Trying 127.0.0.1...

Connected to localhost.

Escape character is '^]'.

Open On-Chip Debugger

At this point the DiscoveryF4 is running, though it may be hung in an error trap. The debugging commands of most interest are--

halt

mdw 0x08000000 16 [‘m’emory ‘d’isplay ‘w’ords; address; number of words]

bp 0x08001234 2 [set ‘b’reak’p’oint; address; the 2 is needed]

bp [display breakpoints]

rbp 0x08001234 [remove breakpoint]

reg [display all the hardware registers]

step [step one instruction]

The general sequence--

With the cursor cut the source code line where a breakpoint is desired

Switch window to the .list file; ctl F (find); and past line into find box

Cut the address (in the .list file) of the instruction to set the breakpoint.

Paste the address into the ‘bp’ command in the terminal window. E.g.--

bp 0x08001234 2

If the F4 is running the breakpoint won’t be set, so ‘halt’ before setting the breakpoint.

After setting the breakpoint, enter ‘reset’ to restart the F4. If the program encounters the breakpoint the debugger stops and it shows the ‘pc’ where it halted. ‘reg’ and ‘mdw’ of the ‘sp’ provide clues as what was going on. Tedious and a last resort.

NOTE: Before recompiling the openocd script that started STLink needs to be killed, i.e. ctl C in the window where it was started. If that is not done the recompiled program is not flashed.

Placing ‘yprintf’ statements along with debugging variables is usually the first place to start. That presumes the program is “mostly working.” A hard fault trap is more difficult. Placing printouts followed by “while(1==1);” to go no further can often isolate the before/after fault location which gives a clue as to where the problem arises. This is done by starting the debugger and entering “halt”. The pc location in the .list file shows where it is trapped.