A. Overview

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.

B. Tasks

1. CanTask01

Location: main Priority: osPriorityIdle (-3)

Generated by: Manual

- Task generated manually (rather than via STM32CubeMX)

- osDelay: 5 seconds

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

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

- Test 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

- Test ‘yscanf’ (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 lines.

4. Task03

Location: main Priority: osPriorityIdle (-3)

Generated by: STM32CubeMX

- Test receiving CAN messages from ‘can\_iface.c’

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

5. CanTxTask

Location: Ourwares/CanTask Priority: 5

Generated by: Manual

Brief: a created queue receives CAN messages from other tasks and places those CAN messages in the ‘can\_driver\_put’ of 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 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 ends with a CR or LF). The code provides for putting lines into a circular buffer for char-by-char mode.

When a line is complete the originating Task waiting for a notification is notified. Only one Task per uart, i.e. there is no provision for distributing lines to multiple Tasks.

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 for that uart. Tasks originating with data to be sent create buffers w buffer control blocks, where the appropriate size is specified. During the endless loop in a task the buffer is loaded and a pointer to the buffer control block created when the buffer was created 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 notify the task originating the buffer just sending that completed and if more items in the circular buffer remain, it starts the sending of next buffer pointed to by the pionted in the circular buffer.

This allows the a task to load a buffer and continue processing while the buffer is queued for sending. Before loading the buffer the task does a ‘NotifyWait, waiting for a notification to task AND the notification bit associated with the particular buffer to be set.

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 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.)

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)

...

#define TSK02BIT00 (1 << 0) // Task notification bit for serial input (SerialTaskReceive.c)

#define TSK02BIT01 (1 << 1) // Task notification bit for serial output buff 21

#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)

#define TSK02BIT04 (1 << 4) // Task notification bit for serial output buff 24

#define TSK02BIT05 (1 << 5) // Task notification bit for serial output buff 25

#define TSK02BIT06 (1 << 6) // Task notification bit for serial output buff 26

...

struct SERIALSENDTASKBCB\* pbuf21 = getserialbuf(&huart6,96,TSK02BIT01,&noteval);

struct SERIALSENDTASKBCB\* pbuf21 = getserialbuf(&huart6,96,TSK02BIT01,&noteval);

struct SERIALSENDTASKBCB\* pbuf24 = getserialbuf(&huart6,24,TSK02BIT04,&noteval);

struct SERIALSENDTASKBCB\* pbuf25 = getserialbuf(&huart6,64,TSK02BIT05,&noteval);

struct SERIALSENDTASKBCB\* pbuf26 = getserialbuf(&huart6,64,TSK02BIT06,&noteval);

...

/\* notification bits except for 'yprintf' output buffer bits \*/

#define TSK02ALLBUT (TSK02BIT00|TSK02BIT02|TSK02BIT03)

/\* A notification copies the internal notification word to this. \*/

uint32\_t noteval = 0; // Receives notification word upon an API notify

/\* notification bits processed after a 'Wait. \*/

uint32\_t noteused = TSK02ALLBUT; // Initially all except 'yprintf' buffers

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 pbuf24 ends up placed on the queue to SerialTaskSend--

yprintf(&pbuf24,"\n\rDMA2 %d",Ddma2-Ddma2\_prev);

2. yprintf.c

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

\* 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 TaskNotifyWait 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 tlhe same as one would use for the standard printf.

The semaphore is used to block other tasks from breaking in during execution of the vsnprintf. Multiple tasks can call yprintf, but will block until printf 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.

The TaskNotifyWait imposes an added limitation when used with multiple notifications not associated with serial output buffers. More on this below. Briefly, the ‘Wait in yprintf might delay the handling of other notifications being implemented for the task, i.e. another notification arrives while yprintf is waiting for the buffer to be released from a prior use.

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \* : Pointer->ret = pointer to CAN control block for this CAN unit

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

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

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

\* 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

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

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’ is 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 bus 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 needs 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, in one (and only one) task the messages are retrieved 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.

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, and the 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 TSK02BIT00 (1 << 0) // Task notification bit for serial input (SerialTaskReceive.c)

#define TSK02BIT01 (1 << 1) // Task notification bit for serial output buff 21

#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)

#define TSK02BIT04 (1 << 4) // Task notification bit for serial output buff 24

#define TSK02BIT05 (1 << 5) // Task notification bit for serial output buff 25

#define TSK02BIT06 (1 << 6) // Task notification bit for serial output buff 26

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, i.e. 21, 24, 25, 26 in the above. With one buffer multiple lines would stall the loop waiting for the buffer to become free and ADC notifications would be lost and/or not and 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 the scheduler places the task in the run mode the ‘noteval’ that receives a copy from the os of the notification word holds the bit or bits giving the notification. 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 one TaskNotifyWait will exit and those bits handled.

Task02 in mxusartusbcan uses multiple notifications along with serial output (yprintf). 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’ buffers are sent the SerialTaskSend interrupt makes notifications. The TaskNotifyWait blocks until a notification is received. The yprintf notifications are handled in the ‘yprintf’ routine, that also includes TaskNotifyWait. The output buffers are marked as released and available for reuse when the notification bit is one. This is because TaskNotify only has the option for Oring in a notification bit, and not ANDing out a bit. To make it work, the buffer bit in the task copy of the notification word has cleared to zero when the buffer is placed in the queue for sending. However, this does not clear the bit in the FreeRTOS internal notification word. Consequently, if other notifications are triggered the internal notification word is copied into the task notification word, and the buffer bit now appears available. If a series of ‘yprintf’ calls are made using the same buffer it is possible that under certain timing conditions the buffer gets written by the a following ‘yprintf’ giving the appearance of having two identical lines with the first one missing. The solution is to use separate buffers, or a different strategy for tasks that have multiple notifications where serial output executed in the course of handling a notification also involves notifications.

D. ‘mxusartusbcan’ initialization

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, 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 to provide floating point output (printf) and input (sscanf). These add substantially to the code size. For sscanf input approximately 18.5K is added to the binary output, less likely to needed.

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

# 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 top 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 line, 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--

Cut source code line where a breakpoint is desired

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

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’ are provide clues as what was going on. Tedious.

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.