Golf Software Driver Use Notes

Version 2.2, August 8, 2019
Burns Fisher, WB1FJ

Note: This document started out describing the Fox drivers. However a few things have changed, generally in minor ways for the Golf RT-IHU. Nonetheless, this document is intended to be only for the RT-IHU, and is checked into the Golf-T IHU repository. The Golf code matching the specifications is currently checked into the Golf-T git repository.

V1.5 of this document updated the SPI driver to include info about supporting multiple device on multiple ports.

V1.6 updates the GPIO to document message-passing capability for interrupts, and defines the intertask message subsystem.

V2.0 Starts becoming more specific to GOLF RT-IHU.

V2.1 Add I2c Golf driver

V2.2 Update the SPI driver to match changes made for the AX5043. General pass through to ensure it is generally correct for all RT-IHU routines.

Serial Line Input/Output

Background and Intentions

Serial line output on the Fox was initially used purely for debugging and testing purposes, not be used in flight. The intent was to be able to connect a PC (or similar) to either the Discovery prototype board (with jumpers and possibly level converters) or to the real IHU board (via the USB port) and to allow the IHU software to respond to commands and reply with text information.

However, in later iterations of the Fox-1 design, it was decided that a serial connection would be available for some experiments to communicate with the IHU. Thus at least some ports will be used in flight, and some changes have been made (as of V1.5 of this document) to transfer blocks of data, potentially using DMA.

On Golf, we are currently only using it for console, but more will be done.

Current State

All routines described in this document are implemented and are working correctly on an N2HET-emulated interface at 38.4K baud only as well as the two hardware UARTs (TI calls them SCI) at any hardware supported speed.

Routines Available

printf, lprintf

Header files required: golf.h, lprintf.h, stdio.h

The lprintf routine here is about what you expect from a printf with a couple exceptions. The most noticeable exception is that it takes \n very literally, and sends a new line character (ASCII lf, decimal 10) without an ASCII cr (decimal 13). So you probably want to specify \n\r where you would normally put \n. The other exception (which I have not tested) is that I suspect it will not do floating point conversions.

Printf is identical to lprintf except that it behaves like printf on a big system regarding \n (i.e. it adds its own \r as well). "Stdout" (the serial port that printf outputs to) is defined by the com port macro DEBUG_PORT.

SerialInitPort

SerialInitPort(COM_NUM comPort, **unsigned int** baud, **unsigned int** qLengthRx,**unsigned int** qLengthTx)

Header files required: golf.h, serial.h

This routine must be called before a serial port is used. In the current implementation, there are definitions for COM1 (the console) COM2 and COM3 that can be used for comPort. Baud is the baud rate; (currently only 38.4K is supported on COM1). It is currently ignored. The "qLength" variables specify how many characters to input or output will be buffered before the task blocks.

SerialGetChar

bool **SerialGetChar**(COM_NUM com, **char** *rxedChar, portTickType timeout)

Header files required: golf.h, serial.h

This routine reads a character from the serial input line specified by 'com'. The character is returned in rxedChar. You can specify a timeout in xBlockTime. If you specify a non-0 number here, the routine will return False if 'timeout' ticks pass without a character being input.

SerialPutString

void SerialPutString(COM_NUM pxPort, **const** portCHAR * **const** pcString, **int length**)

This routine takes a zero-terminated (normal C) string or a buffer and prints it to the specified serial port. If you specify 0 for the string length, the routine assumes a normal C null-terminated string. If you specify a length, it will output 'length' bytes including null bytes.

SerialPutChar

bool SerialPutChar(COM_NUM port, **char** outChar, **portTickType** blockTime)

Header files required: golf.h, serial.h

This routine outputs a single character to the specified COM port. If the output buffer is full, the task will block and only return when the specified character is in the output buffer, or when the blockTime has been exceeded (it will return FALSE in this case).

SPI Input/Output

Background and Intentions

The SPI driver code was developed originally to work with the Ramtron FM25V05 F-RAM chip that Fox initially used. This was later upgraded to a Everspin MR25H40 and later many other devices were added, including the MCP25065 CAN chip, a LTC5599 modulator, gyros etc. It has proven to be fairly general although on Golf, the DCT chip (AX5042) required a slight extension

The SPI code is completely intended to operate as a master, and it is expected that as the master, it initiates all operations by sending a command and 0 or more data bytes, and then awaiting 0 or more response bytes (with an exception, see below).

Current State

Since the first publication of this document, all the SPI routines have been modified to allow specifying a “device” not a “port”. A device refers both to a particular SPI port (or bus) and a specific select line to enable a specific peripheral. For example, as of Fox-1e, the devices are MRAMDev and ModulatorDev.

The code has been tested successfully with several different opcodes for a Ramtron FM25V05 F-RAM chip, the similar MR25H40, and the (quite different) LTC5599 modulator. Only write appears to work on the LTC5599, but that appears to be an oddity of the chip, not the driver.

There is also a completely untested and probably non-functioning time-out routine (which just loops if it gets called).

Routines available

SPIInit

SPIInit(SPIDevice device)

Header files required: spiDriver.h

This routine must be called before using the SPI device specified by “port”, where device is

MRAMDev or DCTDev. More devices will be added. (See Current State above)

SPISendCommand
SPISendCommandBidirectional

bool SPISendCommand(SPIDevice device, uint32_t command, uint16 cmdLen, uint8_t *sndBuf, uint16_t sndLen,uint8_t *rcvBuf, uint16_t rcvLen)

This routine sends ‘cmdLen’ bytes of command specified by 'command' followed by 'sndLen' bytes of data specified in sndBuf, and then keeps clocking the SPI port until rcvLen bytes of data are received back from the slave. The timing of the device may require additional receive bytes to be specified (for example if it takes several clock cycles to start responding to the command). Note that cmdlen, sndLen or RcvLen can be 0. Also, while a command may be only a single byte, you can also use additional bytes in ‘command’ to send additional data. For example, the write routine for the external NVRam (FRAM or MRAM) sends the read command and the address together here, and then specifies the caller’s buffer in sndBuf. The only difference between “command” and “sndbuf” is that command is passed as an immediate value, while sndbuf is passed as an address. Note that one must also be careful about endianness using command.

bool SPIBidirectional(SPIDevice device, uint8_t *sndBuf, uint8_t *rcvBuf, uint16_t length)

This routine sends and receives the same number of bytes simultaneously. It was designed for the AX5043 where the first transmitted bytes is the command and the first received bytes is the status. It is allowed for sndBuf and rcvBuf to be the same buffer, but of course the data originally in sndBuf is overwritten by the data received.

I2c Input/Output

Background and Intentions

The I2c driver code was developed to work with a MAXxxxx thermal measurement chip. It also works with an AX5043 tx/rx chip and should be fine with many other devices.

The I2c code is completely intended to operate as a master.

Current State

This driver has only been used on a simple MAXxxx thermal chip and the AX5043.

Routines available

I2cInit

I2cInit(I2cBus bus)

Header files required: I2cDriver.h

This routine must be called before using the any device on the specified I2c bus.

I2cSendCommand

void I2cSendCommand(I2cBus bus, uint32_t address, uint8_t *sndBuf, uint16_t sndLen,uint8_t *rcvBuf, uint16_t rcvLen)

This routine sends 'sndLen' bytes of data specified in sndBuf, and then receives rcvLen bytes of data from the slave whose address is specified in address.

Either send or receive length can be 0, but that feature has not yet been tried.

1 GPIO Input Output

1.1 Background and Intentions

The GPIO driver is intended to provide a method of manipulating single- or multiple-bit digital input or output. For example, lighting an LED would be single bit output. Monitoring the position of a switch would be single bit input. Reading the 4-bit output from an external command decoder would be multiple-bit digital input. For most of these functions there are no interrupts involved; you specify (a) bit(s) to set/clear or to read, and the change is made or the current value is read.

Some GPIOs can generate interrupts; the interrupt sends an intertask message (see below). The message and the destination task are specified in the GPIO init. The RT-IHU only allows some GPIOs to interrupt (those that are actually designated as GPIOs and not, for example, SPI chip select lines or N2HET pins.

1.2 Current State

Currently, the routines to init, and to read and write single bits and to read multiple bits are all implemented, as are routines to send messages when a GPIO changes.

To add new GPIOs, code is required in the driver, although it is mostly tables and enum additions.

If the code is compiled with DEBUG_BUILD defined, each routine checks to be sure that the bit that has been specified makes sense. For example, if you call GPIOSetOn for a bit that is initialized to be input only, the error routine DebugError is called.

1.3 Routines Available

GPIOInitialize

void GPIOInitialize(Gpio_Use whichGpio,DestinationTask task,
IntertaskMessageType msg, Gpio_Use auxGPIO1);

Required header files: Golf.h, gpio.h, intertask_message.h

This routine must be called for each GPIO bit (or set of bits) that are to be read or set. Gpio_Use is an enum defined in gpio.h which specifies a particular GPIO, for example “WatchdogReset”. The argument whichGPIO specifies which GPIO you are initializing. The code in gpio.c uses tables to understand the details about the specified data bits: which GPIO, which bit, how many bits, input or output, etc.

For GPIOs which will be used to generate interrupts, the driver sends an intertask message when an interrupt is received, along with the values of some other GPIOs. The message consists of a message type and 32 bits of data. GPIOInitialize must specify the destination task, the message type, and one or no GPIOs which to be read and the value sent as the data.

You can also specify NO_TASK for the task, NO_MESSAGE for the message, and NONE for the GPIOs if the GPIO is not to generate an interrupt. For an interrupting GPIO, auxGPIOs can be set to None. If you specify a task or message on a non-interrupting GPIO, an error message is generated (when compiled DEBUG).

GPIOSetOn [glue]

void GPIOSetOn(Gpio_Use whichGpio)

Required header files: Fox.h, gpio.h

Turn the specified bit(s) on.

GPIOSetOff [glue]

void GPIOSetOn(Gpio_Use whichGpio)

Required header files: Fox.h, gpio.h

Turn the specified bit(s) off.

GPIOToggle [glue]

void GPIOToggle(Gpio_Use whichGpio)

Required header files: Fox.h, gpio.h

If the specified bit is on, turn it off. If the specified bit is off, turn it on. This feature only works as described with single bits.

GPIORead [glue]

uint16_t GPIORead(Gpio_Use whichGpio)

Required header files: Fox.h, gpio.h, stdint.h.

Read the current state of the bit(s) specified and return as the value of the function.

4. Non-Volatile Memory

4.1 Background and Intention

The Golf satellites will fly with a non-volatile memory chip variously called FRAM or MRAM (by different manufacturers). The MRAM is highly resistant to radiation damage, and so is useful for long-term data (for example, reset count, historical telemetry, possibly checksum values, whole orbit data, etc.). In addition, the STM32L on Fox contains 4Kb of flash data memory in addition to the 128Kb of program flash memory. The RT-IHU does not have flash memory like this.

Both of these can be useful, so this section describes a common interface to them.

4.2 Current State

These routines are all implemented and working, although the RT-IHU has only a single memory type, ExternalMRAMData.

4.3 Routines Available

All expect “nonvol.h” to be included.

bool writeNV(void const * const data, int dataLength, NVType memoryType, uint32_t address);

This routine writes to the non-volatile memory. The first argument is a pointer to the data, the second is the length in bytes (suitable for ‘sizeof’), the third is an enum that currently includes the types “OnboardData” and “ExternalData”, and address is a byte address from 0 to ‘max address’. Note that

some memory may only read or write in groups of multiple bytes.

Returns TRUE if the write successfully concluded, returns FALSE if it did not (for example if the addresses to be written fall partly or completely outside the range of memory available, or the memory does not exist on this system).

bool readNV(void *data, int dataLength, NVType memoryType, int address);

This routine reads from the non-volatile memory. The arguments are identical to those above.

int getSizeNV(NVType type);

This routine returns the number of bytes (the maximum address +1) in the memory of the type specified.

5. Intertask Messaging

5.1 Background and Intention

The Fox and Golf satellites and derivatives use a real-time operating system which allows a number of different tasks to appear to run independently, although in the same memory space. The OS provides semaphores and queues to allow for synchronization between tasks. A semaphore is simply a flag that allows one task to wait until another task set the flag to say "ok to go"). A queue is similar to a semaphore, but allows one task to give data to another task, and to queue several packets of data in order. Intertask messaging is an organized way to use queues to send messages between tasks. The IM is especially useful since a task can't wait on more than one semaphore. However, it can wait on a single queue while several different tasks put messages in the queue.

5.2 Current State

These routines are all implemented. I have also adopted messaging as the means for an interrupt to communicate with a task.

5.3 Routines Available

All expect "intertask_notify.h" to be included.

void InitInterTask(DestinationTask task, **int** size)
Call this with a destination task and the maximum number of messages that can be queued at one time. The destination task is an enum defined in intertask_notify.h.

bool **WaitInterTask**(DestinationTask task, **int** timeout, Intertask_Message *data)
The receiving task calls this to wait for a message to be sent. It specifies its own task name, a timeout value, and the address of a data structure into which the message will be copied when it is sent. The timeout is specified in hundredths of a second. If WaitInterTask returns true, there is a message in

'data'. If false, the timeout value has been exceeded

bool **NotifyInterTask**(DestinationTask type,**int** timeout, Intertask_Message *data)
The sending task uses this routine to send messages to (and wake up) a destination task. Timeout, specified in hundredth of a second, specifies the time that the sending task will block waiting for the destination task's queue to have room for another message.

bool **NotifyInterTaskFromISR**(DestinationTask type, Intertask_Message *data)
This routine is the same as NotifyInterTask, except that it can be called from an interrupt service routine like a timer or a driver routine. As such it will never block waiting for a message queue to have room. It simply returns false.