Chapter 1

HMT Mini-stack software

Author

Roger Bostock at HMT microelectronics
Daniel Gehriger <gehriger@linkcad.com>

Date

28.01.14

Version

1.0.1

The Mini-stack software is a minimal IO-Link protocol stack implemented with the HMT PHY ICs and an Atmel AT-Mega328P micro controller.

The function of all operating modes of the PHY's (multi-byte, single-byte and transparent), all operating frequencies (COM2 and COM3) and all SIO drive modes (NPN, PNP, Push-pull and Inactive) have been demonstrated with a range of M-sequences with a device tester.

Please refer to the Change History for modifications between the software versions.

This Mini-stack software is primarily designed to demonstrate the function of the HMT7742/HMT7748 PHY's, and to provide a reference design for their use. It *is* intended that users inspect the internals of the code and understand how it functions. The stack is deliberately cut to a minimum number of lines to give users a chance to follow the code. In particular, there is no software support for:

- Events
- ISDU's

The stack is optimised to run efficiently on the microcontroller, and care has been taken to avoid run-time overhead in the interrupt service routines. This demonstrates how the HMT7742/HMT7748 PHY's can be used to reduce the load on the micro-controller (MHz, mA and kBytes)

A demonstration application, see DemoApp, is supplied with the Mini-stack software to facilitate a rapid development start in association with one of the development boards (TM96.1 var. A or B, TM141.0 or TM142.0)

1.1 Stack Usage

Three stack implementations are available, all derived from the templated StackBase base class:

- StackMultiByte: Stack implementation using Multi-Byte mode.
- StackSingleByte: Stack implementation using Single-Byte mode.
- StackTransparent: Stack implementation using Transparent mode.

All implementations share the same API, and a typedef `Stack` is defined as an alias for the selected stack implementation (see Compilation). The stack instance is made available as `Stack::instance`.

The user-provided cyclic code runs inside the application's main loop. It should repeatedly call Stack::instance.canRunUserCode() to determine if it may perform any lengthy calculations or otherwise access the stack. This function will return `true` once per IO-Link communication cycle, **even in the absence of IO-Link communication** when it is set to `true` once per minimum cycle length . When the stack is communicating, this function will run between two sequences, just after the completion of the device response.

Attention

The timer 0 interrupts are important for the stack to track the communication timing, and should not be blocked for more than 50us (max. latency). The timer interrupt routine does not affect the data presented by the stack to the user cyclic code, but may affect the reported operating mode.

Attention

In SIO operation, a PHY data interrupt may arrive at any time and should not be blocked for more than 130us (38.4kBaud operation) or 10us (230.4kBaud operation). The stack is idle in this mode, and will not update or use the process data or write parameter fields returned by Stack::canRunUserCode().

Attention

In established IO-Link operation, the PHY interrupt (only) should be blocked during the operation of the user cyclic code. No data interrupt which requires rapid processing will be received in this time.

Attention

The user cyclic code must complete in the gap between the last device transmission and the end of the master transmission (multi-byte exchange) or between the last device transmission and the start of master transmission (single-byte and transparent exchange)

1.1.1 Compilation

In order to select a stack implementation, include the corresponding stack header file from your application code and compile and link the stack source file.

The code in the stack source files will only be compiled if a corresponding preprocessor symbol has been defined, as shown below. This allows a project file to include all stack implementations and to select the desired stack type by defining the corresponding preprocessor symbol.

- Using the **Multi-Byte** (Io-Link) Mode Stack:
  - compile, and link with, `"stack/stackmultibyte.cpp"` with `STACK_MODE_MULTI_BYTE` defined.
  - include the header file `"stack/stackmultibyte.h"`:

    ```
    #include "stack/stackmultibyte.h"
    ```

- Using the **Single-Byte** Mode Stack:
  - compile, and link with, `"stack/stacksinglebyte.cpp"` with `STACK_MODE_SINGLE_BYTE` defined.
  - include the header file `"stack/stacksinglebyte.h"`:

    ```
    #include "stack/stacksinglebyte.h"
    ```

- Using the **TransparentStack** Mode Stack:
  - compile, and link with, `"stack/stacktransparent.cpp"` with `STACK_MODE_TRANSPARENT` defined.
  - include the header file `"stack/stacktransparent.h"`:

    ```
    #include "stack/stacktransparent.h"
    ```

1.1.2 Hardware configuration for compilation

Multi-byte and single-byte modes are insensitive to the CPU clock frequency, and the internal RC oscillator running at 8MHz is used by default. The define F_CPU=8000000UL should be set to achieve this.

Transparent mode requires an external oscillator running at 18.432MHz and the define F_CPU=18432000UL should be set for this case. The additional define ‘USE_EXT_OSC‘ should be set to enable the correct clock source, which is set by the ATMega fuses.

The correct PHY type, either HMT7742 or HMT7748, must be selected as a preprocessor symbol define. Define either ‘USE_HMT7742‘ to use the HMT7742 PHY, or ‘USE_HMT7748‘ to use the HMT7748 PHY.

The brown-out detector is enabled at the nominal 1.8V level, using the ATMega fuses. The ATmega shows correct behaviour at this level even under conditions where the supply connection has considerable contact bounce.

The ATMega fuse codings are embedded in the source code, and contained in the ".elf" file produced.

1.1.3 Stack Configuration

The selected stack is configured by adjusting the public class constants in its header file, as shown below for the multi-byte mode stack:

```
class StackMultiByte : public StackBase<StackMultiByte,
        1, // PD_IN_SIZE
        1  // PD_OUT_SIZE
>
{
public:
    static const uint8_t REVISION_ID =      IoLink::REVISION_ID_1_1;
    static const uint16_t VENDOR_ID =       0x01a6; // HMT
```

```
    static const uint32_t DEVICE_ID =       0x123457;
    static const uint32_t BAUD_RATE =       230400;
    static const uint8_t MIN_CYC_TIME =     30;  // 30x0.1ms
    static const uint8_t MSEQ_CAPABILITY =  IoLink::MSEQCAP_ISDU_NOT_SUPPORTED
      |
                                            IoLink::MSEQCAP_OP_CODE_0 |
                                            IoLink::MSEQCAP_PREOP_CODE_0;
    static const uint8_t PHY_CFG =          CFG_UVT_16_3V | CFG_RF_ABS |
     CFG_S5V_3_3V
                                  | (BAUD_RATE == 38400 ? CFG_BD_38400 :
     CFG_BD_230400);
    static const uint8_t PHY_CTL_SCT =      CTL_SCT_190MA;
    static const uint8_t PHY_CTL_MODE =     CTL_IOLINK_MODE;
    static const enum SioDriveMode SIO_DRIVE_MODE = DRIVE_MODE_PUSH_PULL;
    static const uint8_t PHY_THERM_DEG =    175; // ~175 degrees Celsius
// [...]
};
```

Most configuration parameters should be obvious and this documentation mentions the reference to the appropriate standard / data sheet documentation.

The **sequence size** and type is determined by the `PD_IN_SIZE` and `PD_OUT_SIZE` template parameters, as well as the class constant `MSEQ_CAPABILITY`, as described in the *"IO-Link Interface and System Specification"*, V1.1.1, section B.1.5.

The stacks all currently implement sequences TYPE_0, TYPE_2_1 through TYPE_2_5 on a Atmega328p with a 8MHz system clock. Larger sequence sizes are also supported at this frequency when using single-byte mode, but other modes may require a higher clock frequency.

All modes both have been tested for the following sequence types at COM2 (38.4kBaud) and COM3 (230.4kBaud):

- TYPE_0
- TYPE_2_1 through TYPE_2_5

Attention

Transparent mode requires an external oscillator running 18.432 MHz and the connection of jumpers on the GENIE Explorer boards to connect the ATMega hardware UART pins TXD (to MOSI) and RXD (to MISO).

Attention

The PHY permanently drives the MISO line in transparent mode, and this interferes with programming of the microcontroller on the SPI interface. For development, a 470ohm resistor may be inserted between the PHY MISO and micro-controller SPI input, to prevent contention when programming. Alternatively delay in startup (setting the PHY to transparent mode) may be added to provide a window to start programming the device, or the debugWIRE may be used for programming.

Attention

Sequence TYPE_1_1/1_2 (interleaved) is not supported

1.1.4 Sample Application

A typical sample application looks like this:

```
#include "stack/stackmultibyte.h"
#include <avr/sleep.h>

void user_configure();
void user_run(const Stack::Parameter* param);

int main(void)
{
    // configure all software modules
    Stack::instance.configure();

    // configure user code
    user_configure();

    // enable interrupts
    sei();

    // select sleep mode
    set_sleep_mode(SLEEP_MODE_IDLE);

    // enter infinite loop: processing is interrupt controlled from now on
    for (;;)
    {
        // enter sleep until interrupt wakes us up
        sleep_mode();

        // check if it's time to run user code
        const Stack::Parameter* paramWrite;
        if (Stack::instance.canRunUserCode(paramWrite))
        {
            Stack::instance.stopInterrupt();
            user_run(paramWrite);
            Stack::instance.restartInterrupt();
        }
    }
}

void user_run(const Stack::Parameter* param)
{
    // check for write access to direct parameter page
    if (param)
    {
        // handle parameter write access
        // [...]
    }
    else if (Stack::instance.stackMode() == Stack::STACK_MODE_SIO)
    {
         // handle SIO mode
         // [...]
    }

    // update process data
    // [...]
}
```

6.3.1 Detailed Description

The DemoApp is a demonstration application which uses the Mini-stack software.

The demonstration application and its related IODD file is supplied with the Mini-stack software to facilitate a rapid development start in assosiation with one of the development boards (TM96.1 var. A or B, TM141.0 or TM142.0)

The related IODD file for this application can be found in directory IODD, files HMT-Mini_stack_38kB-20120727-IOD-D1.0.xml for 38.4kBd variants, and HMT-Mini_stack_230kB-20120727-IODD1.0.xml for 230.4kBd variants.

The demonstration application may be compiled with any stack mode, multi-octet, single-octet or transparent.

The board LEDs indicate the status:

- in SIO-mode the red LED alternately brightens and dims
- in IO-Link operation the green LED alternately brightens and dims
- BUT, the red LED lights permanently if the push-button is pressed.

address	direct parameter	comment
0x10	VendorParamMirrorOutput	value read from device
0x11	VendorParamMirrorInput	value sent to device
0x12	VendorParamPidMode	selects value for process data

Direct Parameter Memory map

Basic data exchange on the direct parameter page is demonstrated. A value written in direct parameter VendorParam-MirrorOutput will subsequently be read in VendorParamMirrorInput.

The process data can have different contents depending on the value set in direct parameter VendorParamPidMode.

setting in VendorParamPidMode	comment	interpretation
0x00	1 bit representing the push-button state	'1' => pressed
0x01	8 bit octet for the potentiometer setting	
0x02	8 bit internally generated saw-tooth value	

Process data selection

6.3.2 Member Function Documentation

6.3.2.1 void DemoApp::run (const **Stack::Parameter** ∗ *param*)

This is the main loop, which is called by the stack after a frame has been received.

Parameters

param	Pointer to Parameter structure if the most recent message completed a write access to the direct parameter page. The data is ∗not∗ automatically written to the direct parameter page, but needs to be manually committed by calling parameterWrite(). This parameter may be NULL if no write access occurred.