ASH WARE Inc.,
I2C eTPU DRIVER
USER MANUAL

WWW.ASHWARE.COM

1. Overview 4
1.1. Revision History 4
1.1.1. Tools Used 4
1.2. References 4
1.3. Definitions 4
2. I2C Master 5
2.1. Hardware Interface and Configuration 5
2.1.1. Pin Configuration 6
2.2. Host Interface 6
2.2.1. Initialization 7
2.2.2. Data Management 8
2.2.2.1. Transfer Request 8
2.2.3. Interrupts 9
2.2.4. Fault Detection 9
2.3. Worst Case Thread Length (WCTL) 9
2.3.1. Latency Requirements 10
2.4. Host Code API Overview 11
3. I2C Slave 12
3.1. Hardware Interface and Configuration 12
3.2. Host Interface 13
3.2.1. Initialization 14
3.2.2. Data Management 15
3.2.3. Idle 15
3.2.4. Interrupts 15
3.2.5. Wait for Data vs. Data Ready Mode 15
3.2.6. Fault Detection 16
3.3. Worst Case Thread Length (WCTL) 16
3.3.1. Latency Requirements 17
3.4. Host API Overview 18
4. I2C Common Build Set 19
4.1. Host-side Utility Software 19

4.2. Build Configuration 19
4.3. Executable Image 19
4.4. Initialized Global Variables 20
4.5. Entry Table Base 20
4.6. MISC Compare Register 20
4.7. Resource Usage 20
4.7.1. Code Memory 20
4.7.2. Data Memory 21

1. Overview

This is the user manual for the Ashware I2C eTPU driver software. This driver provides I2C master and/or slave capability for any Freescale or STMicro MCU with an eTPU or eTPU2 peripheral on-board, with no external hardware required to link into an I2C bus. This manual provides the information necessary to perform system design and software integration with the driver software package. The reader is expected to be familiar with the I2C Bus specification.

1.1. Revision History

Version 1.10 1/6/2016 - update to also support DevTool

Version 1.00 12/1/2011 - initial document

1.1.1. Tools Used

eTPU2+ Development Tool, V2.20A

System Development Tool, V2.20A

ETEC (compiler toolset), V2.42A

eTPU2 Simulator, V4.88A

eTPU2 System Simulator, V4.88A

1.2. References

AMT Publishing: eTPU Programming Made Easy, Munir Bannoura & Margaret Frances

Freescale : ETPURM/D Rev. 1, Enhanced Time Processing Unit (eTPU) Preliminary Reference Manual

The appropriate MCU reference manual, such as *Freescale : MPC5674FRM Rev. 4, MPC5674F Microcontroller Reference Manual*

Phillips Semiconductors: The I2C-Bus Specification, Version 2.1, January 2000

1.3. Definitions

Host Code. The host code is the code that resides on the host CPU and interfaces with the eTPU.

eTPU Code. This is the code that runs in the eTPU.

eTPU Channel. Piece of eTPU hardware, and associated software, that controls one I/O pin.

HSR. Host Service Request – triggers an eTPU processing thread.

SCM. Shared Code Memory. The memory that is initialized at startup with the eTPU executable image.

SDM. Shared Data Memory. The eTPU data memory that can be seen by both the host CPU and the eTPU. Receive channel parameters, data buffers and filter tables are stored in this.

2. I2C Master

The eTPU I2C master driver provides the following set of features:

- up to 400 KHz operation, or better. The actual limit depends upon the eTPU clock rate and other functions/loading of the eTPU.
- read, write and combined format transfers
- 7-bit addressing
- START byte via combined format
- unexpected NACKs reported
- clock stretching (synchronization) by slave devices
- interrupt on transfer completion

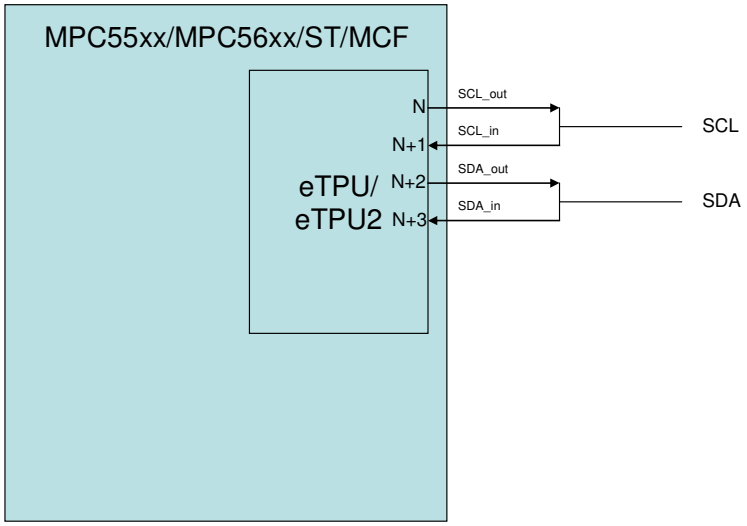
Arbitration (multi-master), high-speed mode, CBUS and 10-bit addressing are not currently supported. The I2C master solution uses 4 consecutive eTPU channels and their associated pins. The first two must be connected to the SCL wire, the second two to the SDA wire.

Note that multiple instances of I2C master and slave drivers can be supported simultaneously, but the bit rate may have to be lowered.

2.1. Hardware Interface and Configuration

The four eTPU channels have the following functions assigned to them, given the base channel number is 'N':

- N + 0 : SCL_out (drives the clock line)
- N + 1 : SCL_in (reads the clock line state for clock stretching / synchronization support)
- N + 2 : SDA_out (drives the data line for write transfers and read acknowledgements)
- N + 3 : SDA_in (reads the data line for read transfers and write acknowledgements)



NOTE: Depending upon how the eTPU module is integrated with a particular MCU, the I2C Master driver may not be able to be mapped to any set of 4 consecutive channels. For example, in some cases a particular eTPU channel is only tied to a pin via its output line, and thus cannot function as an input. Use the appropriate MCU reference when deciding on a system configuration.

2.1.1. Pin Configuration

Pin configuration is accomplished through the System Integration Unit (SIU) Pad Configuration Registers (SIU_PCR). In all cases the PA field must be set appropriately to connect the pin to the eTPU. Input pins for the I2C driver need to the IBE (input buffer enable) field set to 1. Output pins need both the OBE (output buffer enable) field and the ODE (open drain output enable) field set to 1.

2.2. Host Interface

At startup the host must initialize the eTPU module before initializing any I2C drivers. This initialization includes things like loading the eTPU code memory and initialized data memory, and configuring eTPU module registers. It also includes configuration of the two timers in the eTPU, TCR1 and TCR2. The I2C functions base their timing off of the TCR1 counter. Ideally it runs at a frequency at least 100 times higher than the I2C bit rate in order to provide good signal resolution. This code distribution includes eTPU initialization code that can be used as-is or can provide a template. In any case, several outputs of the eTPU software build process are used in the host build and interface:

- etpu_set_scm.h // eTPU code image
- etpu_set_idata.h // eTPU initialized data image
- etpu_set_defines.h // data and function definitions for interfacing to the eTPU

As well as a common eTPU/CPU interface file:

- etpu_i2c_common.h // HSR definitions, error flags, other useful macros

This general initialization is discussed in more detail in section 4. Although initialization is described below, this code package includes host code that handles all the discussed initialization.

2.2.1. Initialization

Initialization of an instance of the I2C master driver is similar to initialization of most other eTPU functions, other than 4 channels are involved. All 4 channels share the same data context, or "channel frame", as it is known. However, each is assigned its own function number. A channel frame should be allocated from eTPU memory and initialized appropriately – the bit timing and setup/hold times, etc. Note that timing configuration is all in terms of TCR1 counts. Data and command buffers should also be allocated at this time; more on that in the next section. The channels are configured with the channel configuration registers (CR) and status control registers (SCR). Before enabling the channels by setting their priority values to a non-zero value the initialization host service requests should be made. Detailed list of channel configuration that must be done for each of the 4 channels that make up the I2C master driver:

- Channel parameter base address (common to all 4 channels). Use the eTPU utility memory allocation routine or other methods to allot a memory chunk from SDM. The size of memory needed is the value of macro `_FRAME_SIZE_I2C_master_` in etpu_set_defines.h
- Function mode. Should be set to 0 on all channels. These are not used by the I2C master.
- Function number. Each eTPU function (entry table entry) has a unique number 0 to 31. Each channel of the I2C driver is associated with a unique entry table. These can be found values can be found in etpu_set_defines.h with the macros `_FUNCTION_NUM_I2C_master_I2C_SCL_out_`, `_FUNCTION_NUM_I2C_master_I2C_SCL_in_`, `_FUNCTION_NUM_I2C_master_I2C_SDA_out_`, and `_FUNCTION_NUM_I2C_master_I2C_SDA_in_`.
- Entry table type. Two types supported, standard or alternate. The setting for each channel (function) can be found in etpu_set_defines.h in the macros `_ENTRY_TABLE_TYPE_I2C_master_I2C_SCL_out_`, `_ENTRY_TABLE_TYPE_I2C_master_I2C_SCL_in_`, `_ENTRY_TABLE_TYPE_I2C_master_I2C_SDA_out_`, and `_ENTRY_TABLE_TYPE_I2C_master_I2C_SDA_in_`.
- Entry table pin direction. Controls which pin helps select the entry vector for servicing an event. The setting for each channel (function) can be found in etpu_set_defines.h in the macros `_ENTRY_TABLE_PIN_DIR_I2C_master_I2C_SCL_out_`, `_ENTRY_TABLE_PIN_DIR_I2C_master_I2C_SCL_in_`, `_ENTRY_TABLE_PIN_DIR_I2C_master_I2C_SDA_out_`, and `_ENTRY_TABLE_PIN_DIR_I2C_master_I2C_SDA_in_`.

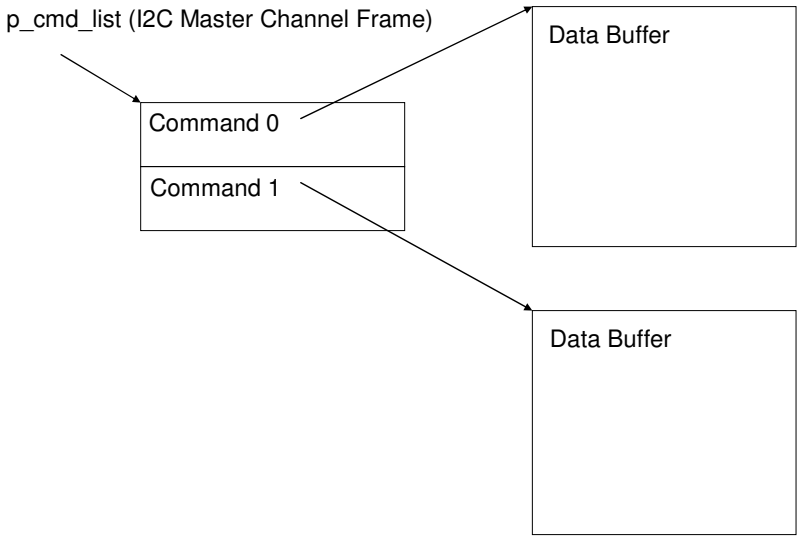
2.2.2. Data Management

The I2C master uses a command list to process message transfers. A buffer in the eTPU data memory must be allocated for this command list. Each command consists of three pieces of data in the following 8-byte structure:

Byte 0	Byte 1	Byte 2	Byte 3
Message Header	Data Buffer Pointer		
0x00	Transfer Size in Bytes		

In most cases a 16-byte buffer to hold two commands is sufficient as that is enough to support a combined format transfer containing two messages.

Next, data buffers for reading or writing must be allocated. There are many ways this could be managed, but for most basic operation allocating two buffers, one for read data and one for write data, is sufficient.



Currently a new transfer and command list cannot be requested and set up until any existing transfer completes (i.e. no transfer queuing).

2.2.2.1. Transfer Request

Host code must first set up the command list for a transfer, and for write transactions fill the appropriate data buffer. Once this is done, a transfer request HSR (`ETPU_I2C_MASTER_START_TRANSFER_HSR - etpu_i2c_common.h`) can be made.

2.2.3. Interrupts

The I2C master issues a channel interrupt from the SCL_out channel once a transfer has completed (or an error has occurred which causes the transfer to end). The interrupt is generated when the STOP is complete (SDA output goes high).

2.2.4. Fault Detection

The I2C master driver can report the following errors (macros defined in etpu_i2c_common.h):

- ETPU_I2C_MASTER_ACK_FAILED – a NACK has been received when an ACK was expected. This could occur because no slave device responds to a requested address, or if a slave device fails to ACK on a written byte. This error flag is not set if a NACK is received on the final byte of a write.
- ETPU_I2C_MASTER_BUSY – a transfer request has been made before the previous request has completed. With proper host software, this error should never occur (the host API in this software package prevents this error).

The error flags can be latched and cleared coherently via host service request (`ETPU_I2C_LATCH_CLEAR_ERRORS_HSR`). Ideally error flags are checked following receipt of a transfer complete interrupt.

2.3. Worst Case Thread Length (WCTL)

The ETEC compiler performs a static analysis to calculate the WCTL for each thread in each eTPU function (class), which are output into the etpu_set_ana.html analysis file. Worst-case RAM accesses per worst case thread are also provided, in case a non-zero RCR (ram collision rate) is to be used in system latency calculations. The WCTL numbers are in "steps", or eTPU instructions. Each one of these requires two eTPU clocks (equivalent to either 2 or 4 system clocks depending upon MCU and system configuration).

Thead/Function	Steps	Ram Accesses
I2C_master	**44**	**23**
I2C_master::I2C_SCL_out	**4**	**0**
I2C_master::InitSCL_out	**4**	**0**
I2C_master::I2C_SCL_in	**4**	**0**
I2C_master::InitSCL_in	**4**	**0**
I2C_master::I2C_SDA_out	**4**	**0**
I2C_master::InitSDA_out	**4**	**0**

I2C_master::I2C_SDA_in	**44**	**23**
I2C_master::InitSDA_in	4	0
I2C_master::Shutdown	2	0
I2C_master::Shutdown	2	0
I2C_master::Shutdown	2	0
I2C_master::Shutdown	2	0
I2C_master::LatchAndClearErrorFlags	3	3
I2C_master::StartTransfer	40	22
I2C_master::PulseClock	35	13
I2C_master::PulseClockIgnore	43	15
I2C_master::ProcessAck	31	9
I2C_master::ProcessAck_Step2	**44**	**23**
I2C_master::ProcessAckIgnore	1	0
I2C_master::BeginStop	12	2
I2C_master::FinishStop	2	1
I2C_master::FinishRepeatedStart	14	4
I2C_master::FinishRepeatedStartIgnore	1	0

The key threads to look at are PulseClock and ProcessAck. These must complete within a half-bit time of their triggering event to completely avoid any latency-induced signal delays.

2.3.1. Latency Requirements

The I2C master eTPU code is resistant to most latency issues. Latency can potentially cause bit times to extend, but operation should remain correct through such an event. In order to ensure proper execution at the specified bit rate, system latency must be less than half of the I2C bit time.

2.4. Host Code API Overview

The section covers the I2C master host software layer that is included with the I2C software distribution. This software provides initialization and transfer request APIs. It does not contain a transfer complete interrupt handler. If the user requires such code for their application, they must write it.

The I2C master host software is contained in the files etpu_i2c_master.h and etpu_i2c_master.c. The .h file should be referenced to get details of all the parameters to the API functions. This software supports multiple master instances, thus every call includes a channel parameter that is the base channel (SCL_out) of the I2C master of interest. This header file also contains the structure definition of transfer commands.

Function Name	Purpose
aw_etpu_i2c_master_init()	Initializes the designated channels to operate as an I2C master. The timing parameters are configured based upon the specified bit rate.
aw_etpu_i2c_master_set_timing()	Allows each individual timing parameter to be set by the user. This interface can be used to override the default timing calculations made by the main initialization routine.
aw_etpu_i2c_master_transmit()	Write the specified number of bytes from the specified data buffer to the specified slave address.
aw_etpu_i2c_master_receive()	Read the specified number of bytes into the specified buffer from the specified slave.
aw_etpu_i2c_master_combined_transfer()	Perform two back-to-back transfers per the specified parameters (combined transfer).
aw_etpu_i2c_master_raw_transfer()	Allows the caller to directly set up a command buffer in order to specify any series of transfers.
aw_etpu_i2c_master_latch_clear_error_flags()	Causes the running error flags to be latched and coherently clears the running flags.
aw_etpu_i2c_master_get_running_error_flags()	Get a snapshot of the running error flags value.
aw_etpu_i2c_master_clear_running_error_flags()	Clear the running error flags.
aw_etpu_i2c_master_get_latched_error_flags()	Get the current latched error flags.

aw_etpu_i2c_master_clear_latched_error_flags()	Clear the latched error flags.

3. I2C Slave

The eTPU I2C slave driver provides the following set of features:

- up to 400 KHz operation, or better. The actual limit depends upon the eTPU clock rate and other functions in the eTPU.
- programmable 7-bit address
- read, write and combined format transfers
- programmable acceptance of general calls
- handles START bytes (ignores the START byte and receives the associated message).
- interrupt on read request and transfer completion
- supports a wait-for-read-data mode wherein the slave driver holds the SCL wire low when a read request is received until the host has filled the read data buffer and alerted that eTPU that the data is ready.

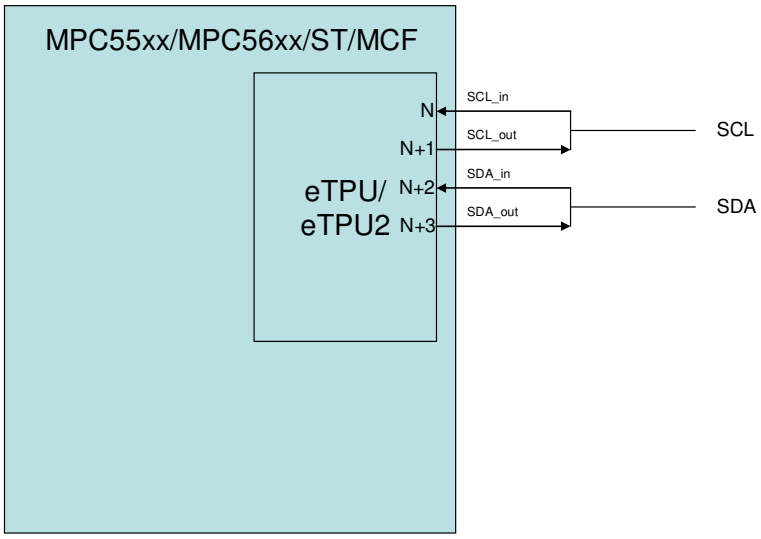
The I2C slave solution uses 4 consecutive eTPU channels and their associated pins. The first two must be connected to the SCL wire, the second two to the SDA wire.

Note that multiple instances of I2C master and slave drivers can be supported simultaneously, but the bit rate may have to be lowered.

3.1. Hardware Interface and Configuration

The four eTPU channels have the following functions assigned to them, given the base channel number is 'N':

- N + 0 : SCL_in (reads the clock line state for timing of bit reads and writes)
- N + 1 : SCL_out (drives when clock line when waiting for the host to supply read data, thereby holding off the master)
- N + 2 : SDA_in (reads the data line for write transfers and read acknowledgements)
- N + 3 : SDA_out (drives the data line for read transfers and write acknowledgements)



NOTE: Depending upon how the eTPU module is integrated with a particular MCU, the I2C slave driver may not be able to be mapped to any set of 4 consecutive channels. For example, in some cases a particular eTPU channel is only tied to a pin via its output line, and thus cannot function as an input. Use the appropriate MCU reference when deciding on a system configuration.

See section 2.1.1 for information regarding pin configuration.

3.2. Host Interface

At startup the host must initialize the eTPU module before initializing any I2C drivers. This initialization includes things like loading the eTPU code memory and initialized data memory, and configuring eTPU module registers. It also includes configuration of the two timers in the eTPU, TCR1 and TCR2. The I2C functions base their timing off of the TCR1 counter. Ideally it runs at a frequency at least 100 times higher than the I2C bit rate in order to provide good signal resolution. This code distribution includes eTPU initialization code that can be used as-is or can provide a template. In any case, several outputs of the eTPU software build process are used in the host build and interface:

- etpu_set_scm.h // eTPU code image
- etpu_set_idata.h // eTPU initialized data image
- etpu_set_defines.h // data and function definitions for interfacing to the eTPU

As well as a common eTPU/CPU interface file:

- etpu_i2c_common.h // HSR definitions, error flags, other useful macros

This general initialization is discussed in more detail in section 4. Although initialization is described below, this code package includes host code that handles all the discussed initialization.

3.2.1. Initialization

Initialization of an instance of the I2C slave driver is similar to initialization of most other eTPU functions, other than 4 channels are involved. All 4 channels share the same data context, or "channel frame", as it is known. However, each is assigned its own function number. A channel frame should be allocated from eTPU memory and initialized appropriately – address, setup/hold times, etc. Note that timing configuration is all in terms of TCR1 counts. Data buffers should also be allocated at this time; more on that in the next section. The channels are configured with the channel configuration registers (CR) and status control registers (SCR). Before enabling the channels by setting their priority values to a non-zero value the initialization host service requests should be made. Detailed list of channel configuration that must be done for each of the 4 channels that make up the I2C slave driver:

- Channel parameter base address (common to all 4 channels). Use the eTPU utility memory allocation routine or other methods to allot a memory chunk from SDM. The size of memory needed is the value of macro `_FRAME_SIZE_I2C_slave_` in etpu_set_defines.h
- Function mode. Only the SCL_in channel of the I2C slave driver uses function mode, and it only looks at bit 0. If FM0 is set to 0 ("data ready mode") then the software assumes the read buffer has been filled and starts outputting data in response to a read request without delay. If FM0 is set to 1 ("wait for data mode") then the slave driver will hold the SCL line low until the host has acknowledged that it has filled the read buffer (via HSR `ETPU_I2C_SLAVE_DATA_READY`). The FM bits should be set to 0 on all other channels.
- Function number. Each eTPU function (entry table entry) has a unique number 0 to 31. Each channel of the I2C driver is associated with a unique entry table. These can be found values can be found in etpu_set_defines.h with the macros `_FUNCTION_NUM_I2C_slave_I2C_SCL_out_`, `_FUNCTION_NUM_I2C_slave_I2C_SCL_in_`, `_FUNCTION_NUM_I2C_slave_I2C_SDA_out_`, and `_FUNCTION_NUM_I2C_slave_I2C_SDA_in_`.
- Entry table type. Two types supported, standard or alternate. The setting for each channel (function) can be found in etpu_set_defines.h in the macros `_ENTRY_TABLE_TYPE_I2C_slave_I2C_SCL_out_`, `_ENTRY_TABLE_TYPE_I2C_slave_I2C_SCL_in_`, `_ENTRY_TABLE_TYPE_I2C_slave_I2C_SDA_out_`, and `_ENTRY_TABLE_TYPE_I2C_slave_I2C_SDA_in_`.
- Entry table pin direction. Controls which pin helps select the entry vector for servicing an event. The setting for each channel (function) can be found in etpu_set_defines.h in the macros `_ENTRY_TABLE_PIN_DIR_I2C_slave_I2C_SCL_out_`, `_ENTRY_TABLE_PIN_DIR_I2C_slave_I2C_SCL_in_`, `_ENTRY_TABLE_PIN_DIR_I2C_slave_I2C_SDA_out_`, and `_ENTRY_TABLE_PIN_DIR_I2C_slave_I2C_SDA_in_`.

3.2.2. Data Management

The channel frame of the I2C slave contains a pointer to a write buffer and a read buffer. At initialization time these should be configured. These pointers could be changed on the fly, but this would have to be done very carefully to avoid doing so mid-transfer. The read buffer (read from master perspective; slave sends data out from this buffer when commanded by the master to do so) could be switched when using the "wait for data" mode. The write buffer should be read upon receipt of a transfer complete interrupt to avoid having it overwritten by an ensuing write transfer. Both the read and write buffer have a size associated with them. The driver will not read or write past the designated size, and throws an error if the transfer causes such a case. Note that on a write transaction, if the write buffer end is reached the slave driver continues to issue an ACK rather than a NACK, but the data is thrown away. A fault flag is set – see 3.2.5.

3.2.3. Idle

When the SDA and SCL lines are high for at least t_{BUF} time the slave goes into idle mode, awaiting a falling transition on the SDA line – the start of a new message transfer. When such a transfer is detected but not addressed for this slave, the slave goes into a search for idle or a new (repeated) START. During this time, depending upon the t_{BUF} setting, spurious invalid start faults can be detected.

3.2.4. Interrupts

The I2C slave driver issues two different channel interrupts. Once is sourced from the SCL_in channel. This is issued when a read command header byte has been received targeting this slave and the "wait for data" mode is enabled. When using the "wait for data" mode the host should respond to this interrupt by preparing the read data buffer and then issuing the data ready HSR (`ETPU_I2C_SLAVE_DATA_READY`) – see 3.2.5 for more detail. The SCL_in channel will also generate an interrupt when an invalid or unexpected STOP occurs. The error flags should be checked when this interrupt is handled.

The other interrupt is sourced from the SDA_in channel. It is issued when a transfer has completed (read or write). The interrupt is set on receipt of a STOP, or a repeated START.

3.2.5. Wait for Data vs. Data Ready Mode

The I2C slave can be operated in two different modes in terms of how it handles read transfer requests from a master. In "data ready" mode the data buffer that is read by the master must be configured before such a transfer occurs. The I2C slave software will start outputting the bytes from the read buffer without interrupting the host processor.

On the other hand, when the "wait for data" mode is enabled, the slave eTPU code will interrupt the host processor after the ACK bit of the header byte, and simultaneously hold the SCL line low. The host must respond to the interrupt and fill and configure a read buffer (if not already done), and then alert the eTPU I2C slave driver that the read buffer is ready via an HSR (`ETPU_I2C_SLAVE_DATA_READY`). When the slave processes the host service request it starts outputting the data bits for the master to read and releases the

SCL line. Note that the interrupt handler can read the received header byte if needed for use in configuring the read buffer.

3.2.6. Fault Detection

The I2C slave driver can report the following errors:

- ETPU_I2C_SLAVE_INVALID_START – The transition from an idle bus (SDA and SCL high) to START (falling SDA edge followed by falling SCL edge) did not occur as expected. Signals on the bus are ignored until idle is detected again.
- ETPU_I2C_SLAVE_BUFFER_OVERFLOW – A read or write transfer exceeded the specified buffer size. Message processing continues but written bytes are ignored and read bytes all go out as 0x00.
- ETPU_I2C_SLAVE_STOP_FAILED – A STOP was expected (NACK received on read transfer), but was not detected, or an improperly formed STOP was detected. Further signal transitions are ignored until idle is detected.

The error flags can be latched and cleared coherently via host service request (`ETPU_I2C_LATCH_CLEAR_ERRORS_HSR`). Ideally error flags are checked following receipt of a transfer complete interrupt.

3.3. Worst Case Thread Length (WCTL)

The ETEC compiler performs a static analysis to calculate the WCTL for each thread in each eTPU function (class), which are output into the etpu_set_ana.html analysis file. Worst-case RAM accesses per worst case thread are also provided, in case a non-zero RCR (ram collision rate) is to be used in system latency calculations. The WCTL numbers are in “steps”, or eTPU instructions. Each one of these requires two eTPU clocks (equivalent to either 2 or 4 system clocks depending upon MCU and system configuration).

Thead/Function	Steps	Ram Accesses
I2C_slave	**73**	**20**
I2C_slave::I2C_SCL_in	**6**	**3**
I2C_slave::InitSCL_in	**6**	**3**
I2C_slave::I2C_SCL_out	**4**	**0**
I2C_slave::InitSCL_out	**4**	**0**
I2C_slave::I2C_SDA_in	**6**	**3**
I2C_slave::InitSDA_in	**6**	**3**

I2C_slave::I2C_SDA_out	**73**	**20**
I2C_slave::InitSDA_out	4	0
I2C_slave::Shutdown	2	0
I2C_slave::Shutdown	2	0
I2C_slave::Shutdown	2	0
I2C_slave::Shutdown	2	0
I2C_slave::ReadDataReady	40	13
I2C_slave::LatchAndClearErrorFlags	3	3
I2C_slave::IdleDetectPass_SDA	13	3
I2C_slave::IdleDetectPass_SCL	11	3
I2C_slave::IdleDetectFail_SDA	9	3
I2C_slave::IdleDetectFail_SCL	9	3
I2C_slave::TransferStart_SDA	16	3
I2C_slave::TransferStart_SCL	18	5
I2C_slave::DataBitReady	51	11
I2C_slave::OutputDataBit	22	6
I2C_slave::HandleAck	**73**	**20**
I2C_slave::FoundStop	19	6
I2C_slave::FoundRepeatedStart	19	6

The key threads are DataBitReady, OutputDataBit and HandleAck. All of these threads need to run within one bit time, and they need to start with a delay of less than half a bit time for proper operation.

3.3.1. Latency Requirements

The I2C slave is more latency sensitive than the master. This is because it is the recipient of the signals rather than controlling them, so in some cases if it misses the window to read a pin or receive a signal, the transfer will fail. For example, the I2C slave must

service the SDA input high-low transition before the SCL pin transitions to low in order to detect a (repeated) START.

3.4. Host API Overview

The section covers the I2C slave host software layer that is included with the I2C software distribution. This software provides initialization and APIs to read received data and set up data to be transmitted to requesting masters. It does not contains any interrupt handlers (transfer complete, or wait for data) - if the user requires such code for their application, they must write it.

The I2C slave host software is contained in the files etpu_i2c_slave.h and etpu_i2c_slave.c. The .h file should be referenced to get details of all the parameters to the API functions. This software supports multiple slave instances, thus every call includes a channel parameter that is the base channel (SCL_in) of the I2C slave of interest.

Function Name	Purpose
aw_etpu_i2c_slave_init()	Initializes the designated channels to operate as an I2C slave.
aw_etpu_i2c_slave_set_read_buffer()	Sets the read buffer pointer and (maximum) size.
aw_etpu_i2c_slave_issue_data_ready()	Issues a host service request to the slave to indicate the read data buffer is ready for use in the pending read transfer.
aw_etpu_i2c_slave_set_write_buffer()	Sets the write buffer and its maximum size.
aw_etpu_i2c_slave_get_transfer_status()	Get the header, transfer byte count and error flags. Meant to be used after a transfer completes (and interrupt is set).
aw_etpu_i2c_slave_get_write_data()	API to retrieve data written to the slave by a master. It retrieves the data into the specified buffer (should be in host memory) and also returns the header byte and size. Note the destination buffer should be sized to be at least as large as the slave write buffer in the eTPU, to avoid the chance of an overrun.
aw_etpu_i2c_slave_latch_clear_error_flags()	Causes the running error flags to be latched and coherently clears the running flags.
aw_etpu_i2c_slave_get_running_error_flags()	Get a snapshot of the running error flags

	value.
aw_etpu_i2c_slave_clear_running_error_flags()	Clear the running error flags.
aw_etpu_i2c_slave_get_latched_error_flags()	Get the current latched error flags.
aw_etpu_i2c_slave_clear_latched_error_flags()	Clear the latched error flags.

4. I2C Common Build Set

The I2C source code can be included with other eTPU code in order to support additional eTPU functionality. The I2C driver also comes pre-built, so if no other eTPU functions are needed all components for integration are ready right out of the package. This section is mainly applicable when using the pre-built components, but the same concepts apply if the I2C software is combined with other eTPU code into a larger build.

Several auto-generated outputs of the eTPU software build process are used in the host build and interface:

- etpu_set_scm.[c,h] // eTPU code image
- etpu_set_idata.[c,h] // eTPU initialized data image
- etpu_set_defines.h // data and function definitions for interfacing to the eTPU

As well as a common eTPU/CPU interface file:

- etpu_i2c_common.h // HSR definitions, error flags, other useful macros

4.1. Host-side Utility Software

Included in this package are several eTPU utility files based upon the Freescale eTPU software distribution. These include:

- etpu_init.[c,h]
- etpu_util.[c,h]
- etpu_struct.h

These provide an eTPU register mapping structure, eTPU initialization routines and functions to allocate and read/write eTPU memory. The distribution also includes a set of processor-specific memory-map structure header files for the common MPC processors with an eTPU/eTPU2 module onboard.

4.2. Build Configuration

There are no build-time configuration options for the I2C software at this time.

4.3. Executable Image

The eTPU executable code image is contained in etpu_set_scm.h as a set of opcodes. This header file can be included in host source code to produce an initialized array, or

etpu_set_scm.c can be added to the host build – it defines an initialized array that represents the eTPU executable image.

At power-up, this array must be block copied to the eTPU's SCM memory by the host code as part of the eTPU's initialization process.

4.4. Initialized Global Variables

eTPU code can have initialized global variables. I2C does not have any, although the built-in global error handling library includes one 32-bit error status word. Like the executable code image, the standard procedure is to place this eTPU data in an in initialized array, which is then copied to the eTPU SDM as part of the startup initialization process. The file etpu_set_idata.h contains the data; etpu_set_idata.c can be used directly as it defines an initialized array of the eTPU data.

4.5. Entry Table Base

When eTPU code is compiled, an entry table is generated and located. By default, the entry table is located at SCM address 0x0000, but if for some reason the build is configured differently, it may be located at a non-zero address. The eTPU's ECR register contains a field named ETB that establishes the Event Vector Table's (entry table) base address. The host code must initialize this field to correspond to the base address of the entry table.

The `_ENTRY_TABLE_BASE_ADDR_` #define specifies the Event Vector Tables (entry tables) base address. It is found in the etpu_set_defines.h file.

4.6. MISC Compare Register

The eTPU performs a continuous checksum of the eTPU's code memory. This is compared for validity against the ETPUMISCCMPR register. The host code must initialize this register with the correct checksum value.

The `_MISC_VALUE_` #define specifies the checksum's valid value that should be written to the MISC register. It is found in the etpu_set_defines.h file.

4.7. Resource Usage

The sections below detail the amount of code (SCM) and data (SDM) memory consumed by the most current version of the I2C master/slave drivers. Note that dynamically allocated data buffers for handling transfers are not included, as these are configured by the user.

4.7.1. Code Memory

The amount of executable eTPU code in the master and slave drivers is as follows:

File	Opcodes	Code Size (bytes)	Start	Stop
etec_i2c_slave.c	322	1288	0xC80	0x1184

etec_i2c_master.c	288	1152	0x800	0xC7C

Also, since each I2C driver uses 4 entry table, and each entry table requires 64 bytes, another (8 * 0x40) = 0x200 (512) bytes are used for entry table support.

4.7.2. Data Memory

The channel frame sizes for an I2C master and slave instance are shown below.

```
#define _FRAME_SIZE_I2C_master_                 0x40
#define _FRAME_SIZE_I2C_slave_                  0x40
```

Additional data space must be allocated for command (master only), write and read buffers.