AN12149

Implementing an IEEE 1588 V2 on i.MX RT Using PTPd, FreeRTOS, and lwIP TCP/IP Stack

Rev. 2.0 — 31 March 2025 Application note

Document information

Information	Content
Keywords	AN12149, FreeRTOS, 1588 PTPd Clock synchronization
Abstract	This application note describes the implementation of the IEEE 1588 V2 Precision Time Protocol (PTP) on the i.MX RT MCUs running FreeRTOS.

1 Introduction

This application note describes the implementation of the IEEE 1588 V2 Precision Time Protocol (PTP) on the i.MX RT MCUs running FreeRTOS. The IEEE 1588 standard provides accurate clock synchronization for distributed control nodes for industrial automation applications.

The implementation runs on the i.MX RT10xx Evaluation Kit (EVK) board with i.MX RT10xx MCUs. The demo software is based on the NXP MCUXpresso SDK 2.15.x for i.MX RT10xx EVK boards. The demo is a PTP daemon (PTPd) using the lwIP TCP/IP stack shipped with the MCUXpresso SDK IDE and runs on the FreeRTOS. PTPd is an open source implementation of the PTP.

This document describes the IEEE 1588 protocol basics, the IEEE 1588 functions on i.MX RT10xx MCUs, and the detailed description of the IEEE 1588 demo software. It includes how to port PTPd for Amazon FreeRTOS on i.MX RT10xx MCUs and how to enable the ENET output compare function to monitor the clock synchronization status. This document also describes how to build and run the demo.

2 IEEE 1588 basic overview

The IEEE 1588 standard is known as the Precision Clock Synchronization Protocol for Networked Measurement Control Systems, also known as Precision Time Protocol (PTP). The IEEE 1588 PTP enables the clocks to be distributed across an Ethernet network and accurately synchronized using a process where the distributed nodes exchange timestamped messages.

The technology of the standard was originally developed by Agilent Technologies, Inc. and is used for distributed measuring and control tasks. The challenge is to synchronize the networked measuring devices with each other in terms of time, making them able to record measured values and providing them with a precise system timestamp. Based on this timestamp, the measured values can then be correlated with each other.

Typical applications of the IEEE 1588 time synchronization include:

- Time-sensitive telecommunication services that require precise time synchronization between communicating nodes.
- Industrial network switches that synchronize sensors and actuators over a single-wire distributed control network to control automated assembly processes.
- Powerline networks that synchronize across large-scale distributed power grid switches to enable smooth transfer of power.
- Test/measurement devices that must maintain accurate time synchronization with the device under test in many different operating environments.
- Printing machines, cooperative robotic systems, and residential Ethernet.

These applications require precise clock synchronization between the devices with accuracy in the sub-microsecond range. It is a remarkable feature of IEEE 1588 that this synchronization precision is achieved through regular Ethernet connectivity with standard Ethernet frames.

This solution enables nearly any device of any performance to participate in high precision synchronized networks that are simple to operate and configure.

Other key benefits of the IEEE 1588 protocol include:

- Convergence times of less than a minute for sub-microsecond synchronization between heterogeneous distributed devices with different clocks, resolution, and stability.
- Automatic configuration and segmentation. Each node uses the Best Master Clock (BMC) algorithm to determine the best clock in the segment. Every PTP node stores its features within a specified dataset. These features are transmitted to other nodes within sync telegrams. Based on this, the other nodes are able to synchronize their data sets with the features of the actual master and can adjust their clocks. The cyclic

running of the BMC also allows hot swapping; that is, nodes can be connected or removed during propagation time.

- Simple configuration and operation with low computing resource requirements and network bandwidth consumption.

2.1 Synchronization principle

Network clocks are organized in a master-slave hierarchy. IEEE 1588 identifies the master clock and then establishes a two-way timing exchange by which the master sends messages to its slaves to initiate synchronization. Each slave then responds to synchronize itself to its master. This sequence is repeated throughout the specified network to achieve and maintain clock synchronization.

The process starts with one node (master clock) transmitting a sync telegram that contains the estimated transmission time. The exact transmission time of the sync telegram is captured by a clock and transmitted in a second follow-up message. By comparing the timestamp information contained within the first and second telegrams against its own clock, the receiver can calculate the time difference between its own clock and the master clock (as shown in Figure 1). The sync and follow-up messages are sent as a multicast. Some IEEE 1588 systems enable hardware timestamping and the insertion of actual timestamps into the sync messages. In this case, the follow-up messages are not needed (one-step mode of operation).

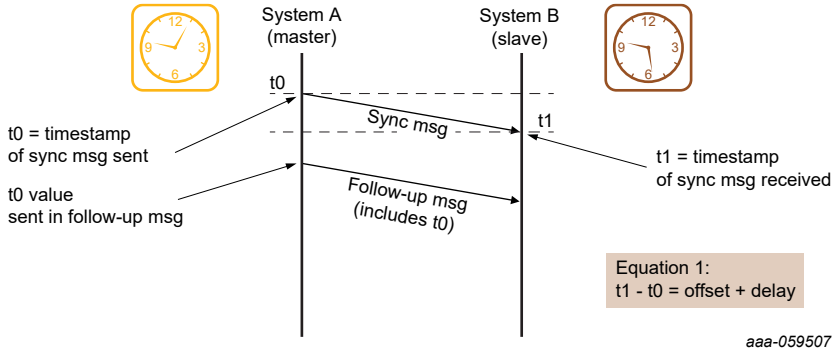

Figure 1. Offset and delay measurement - Sync message, follow-up message

The telegram propagation time is determined cyclically in a second transmission process between the slave and the master (delay telegrams). The slave can then adjust its clock and adapt it to the current bus propagation time (as shown in Figure 2). The `delay_req` and `delay_resp` messages are point-to-point, but sent with a multicast address for simplicity reasons.

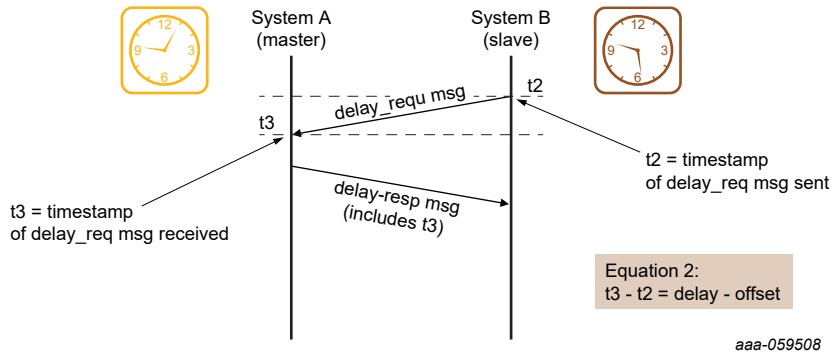

Figure 2. Offset and delay measurement - Delay messages

Figure 3 shows an example of the IEEE 1588 synchronization sequence (one cycle) and the calculation of the actual offset and delay between the master and slave nodes.

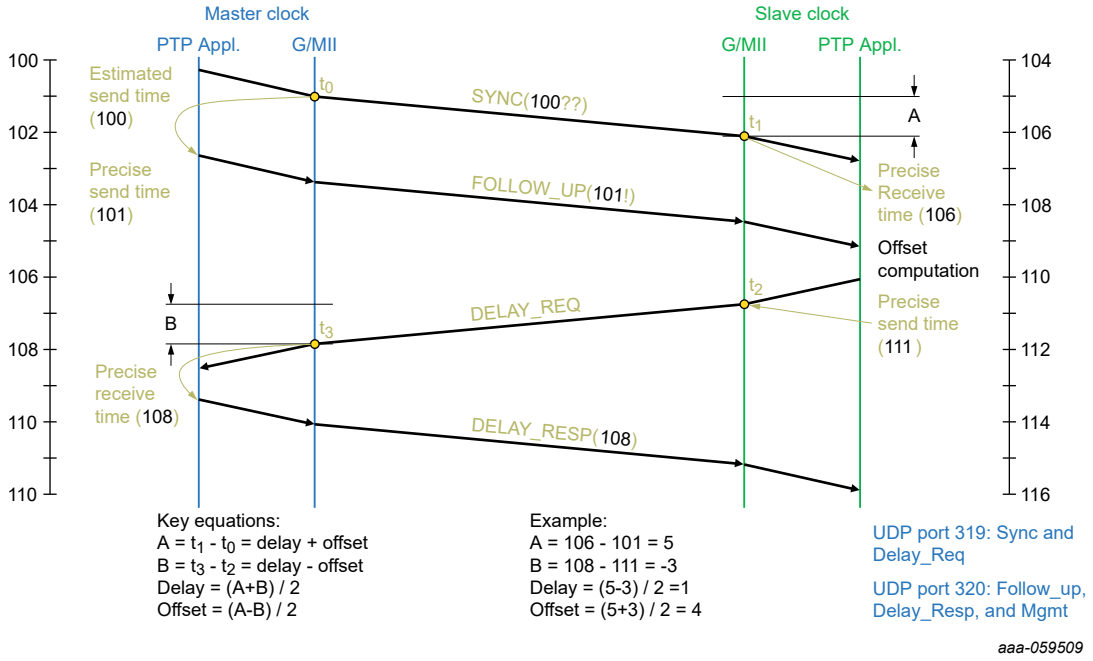

Figure 3. IEEE 1588 synchronization message sequence

For more information about the IEEE 1588 standard, visit the National Institute of Standards and Technology.

2.2 Timestamping

The PTP protocol can be completely implemented into the software using a standard Ethernet module. Because the timestamp information is applied at the application level, the delay fluctuation introduced by the software stack running on both the master and slave devices means that only a limited precision can be achieved (as shown in Figure 4).

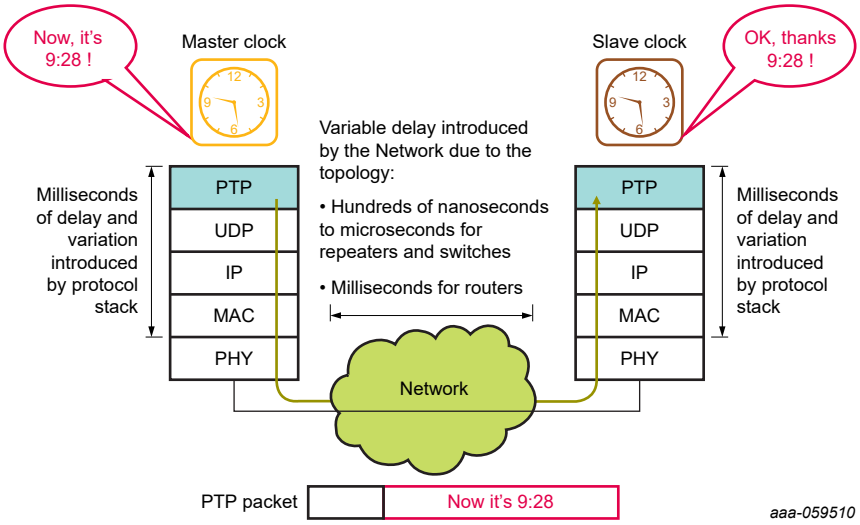

Figure 4. Software timestamp implementation

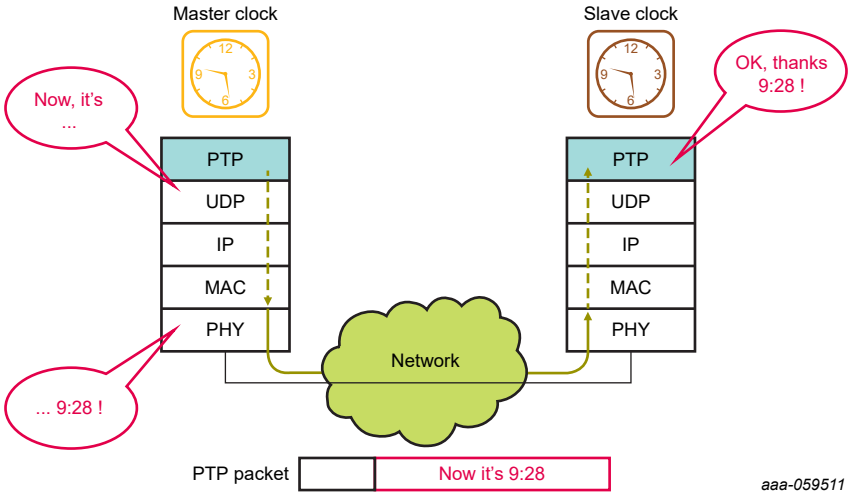

Figure 5. Hardware timestamp implementation

It is possible to minimize the impact of the protocol stack delay by taking timestamps closer to the physical interface, that is, at the MAC or PHY layers (as shown in Figure 5). A dedicated hardware with timestamping capabilities (such as the MAC-NET peripheral module or the 10/100-Mbps Ethernet MAC (ENET) of the NXP i.MX RT 1050 and 1020) enables synchronization with significantly improved accuracy.

3 IEEE 1588 functions on i.MX RT

The i.MX RT10xx devices integrate the MAC-NET core (in conjunction with a 10/100-Mbit/s MAC) to accelerate the processing of various common networking protocols, such as IP, TCP, UDP, and ICMP, providing wire speed services to client applications. The unified DMA (uDMA), internal to the ENET module, optimizes data transfer between the ENET core and the SoC and supports the enhanced buffer descriptor programming model to support IEEE 1588 functionality. To enable IEEE 1588 (or similar) time synchronization protocol implementations, the MAC is combined with a timestamping module to support precise timestamping of incoming and outgoing frames. To enable the 1588 support, set the EN1588 bit in the `ENET_ECR` (Ethernet Control Register).

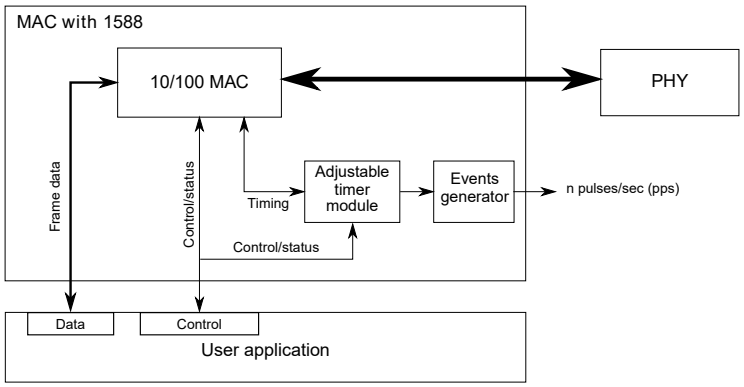

Figure 6. IEEE 1588 functions overview

3.1 Adjustable timer module

The Adjustable Timer Module (TSM) implements the Free-Running Counter (FRC), which generates the timestamps. The FRC operates with the time-stamping clock, which can be set to any value, depending on your system requirements.

Through a dedicated correction logic, the timer can be adjusted to enable synchronization with a remote master and provide synchronized timing reference to the local system. The timer can be configured to trigger an interrupt after a fixed time period to allow synchronization of software timers or perform other synchronized system functions.

The timer is typically used to implement a period of one second; hence, its value ranges from 0 to $(1 \times 10^9) - 1$. The period event can trigger an interrupt and the software can maintain the seconds' and hours' time values as necessary.

The adjustable timer consists of a programmable counter/accumulator and a correction counter. The periods of both counters and their increment rates are freely configurable, allowing for a very fine tuning of the timer.

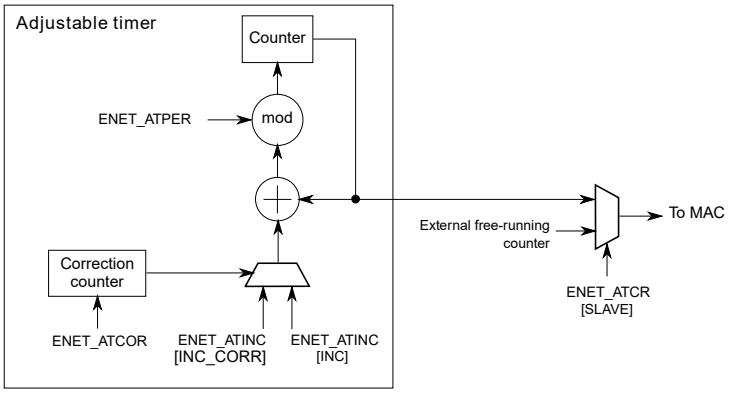

Figure 7. Adjustable timer implementation detail

The counter provides the current time. During each timestamping clock cycle, a constant value is added to the current time, as programmed in `ENET_ATINC` (Timestamping Clock Period Register). The value depends on the selected timestamping clock frequency. For example, if it operates at 125 MHz, setting the increment to eight represents 8 ns.

The period, configured in `ENET_ATPER` (Timer Period Register), defines the modulo when the counter wraps. In a typical implementation, the period is set to 1×10^9 so that the counter wraps every second. All timestamps represent the absolute nanoseconds within a period of 1 ns. When this period is reached, the counter wraps to start again, respecting the period modulo. This means it does not necessarily start from zero, but the counter is loaded with the value (Current + Inc – (1×10^9)), assuming the period is set to 1×10^9.

The correction counter is completely independent and increments by one with each timestamping clock cycle. When the counter reaches the value configured in `ENET_ATCOR` (Timer Correction Register), it restarts and instructs the timer to increment by the correction value once, instead of the normal value.

The normal and correction increments are configured in `ENET_ATINC`. To speed up the timer, set the correction increment higher than the normal increment value. To slow the timer down, set the correction increment lower than the normal increment value.

The correction counter defines only the distance of the corrective actions, not the amount. This allows for very fine corrections and low jitter (in the range of 1 ns), independent of the selected clock frequency.

3.2 Transmit timestamping

Only 1588 event frames must be timestamped on transmit. The client application (for example, the MAC driver) shall detect 1588 event frames and set the TS bit in the TxBD (Enhanced Transmit Buffer Descriptor) together with the frame.

If `TxBD[TS]` is set, the MAC records the timestamp for the frame in `ENET_ATSTMP` (Timestamp of Last Transmitted Frame Register). The `TS_AVAIL` bit in `ENET_EIR` (Interrupt Event Register) is set to indicate that a new timestamp is available.

The software implements a handshaking procedure by setting TxBD[TS] when it transmits the frame for which a timestamp is needed and waits for ENET_EIR[TS_AVAIL] to determine when the timestamp is available. The timestamp is then read from the ENET_ATSTMP register. This is done for all event frames. Other frames do not use TxBD[TS] and do not interfere with the timestamp capture.

3.3 Receive timestamping

When a frame is received, the MAC latches the value of the timer when the frame's start of frame delimiter (SFD) field is detected, and provides the captured timestamp in the 1588 timestamp field defined in the RxBD (Enhanced uDMA receive buffer descriptor). This is done for all received frames.

3.4 Time synchronization

The adjustable timer module is available to synchronize the local clock of a node to a remote master. It implements a free-running 32-bit counter and also contains an additional correction counter.

The correction counter increases or decreases the rate of the free-running counter, enabling very fine granular changes of the timer for synchronization, yet adding only a very low jitter when performing corrections.

The application software implements the required control algorithm (in the slave scenario), setting the correction to compensate for local oscillator drifts and locking the timer to the remote master clock on the network.

The timer and all timestamp-related information should be configured to show the true nanoseconds value of one second (the timer is configured to have a period of one second). Hence, the values range from 0 to (1×10^9) - 1. In this application, the seconds counter is implemented in software using an interrupt function that is executed when the nanoseconds counter wraps at 1×10^9.

3.5 Input capture and output compare

The input capture and output compare block can be used to provide precise hardware timing for input and output events. The IEEE 1588 timer has four channels. Each channel supports input capture and output compare using the 1588 counter.

In the input capture mode, the TCCRn (Timer Compare Capture Register, n = 1, 2, 3, 4) latches the time value when the corresponding external event occurs. An event can be the rising, falling, or either edge of one of the 1588_TMRn signals. An event causes the corresponding TCSRn[TF] (Timer Control Status Register) timer flag to be set, indicating that an input capture occurred. If the corresponding interrupt is enabled with the TCSRn[TIE] field, an interrupt can be generated.

In the output compare mode, the TCCRn compare registers are loaded with the time at which the corresponding event shall occur. When the ENET free-running counter value matches the output compare reference value in the TCCRn register, the corresponding flag (TCSRn[TF]) is set, indicating that an output compare occurred. The corresponding interrupt (if enabled by TCSRn[TIE]) is generated. The corresponding 1588_TMRn output signal is asserted according to TCSRn[TMODE].

4 IEEE 1588 implementation for i.MX RT

The MIMXRT10xx Evaluation Kit (EVK) board is used as the hardware platform for hardware timestamping-based IEEE 1588 V2 PTP. The solution uses the MCUXpresso SDK IDE for the i.MX RT10xx EVK board, which includes the NXP ENET driver of the i.MX RT10xx MCU, the PHY driver, the ported FreeRTOS, and the ported lwIP TCP/IP stack for the EVK-MIMXRT10xx board. The IEEE1588 V2 PTP is implemented by the PTP daemon application, which is an open-source implementation of the PTP. Figure 8 shows the hardware and software components of this solution.

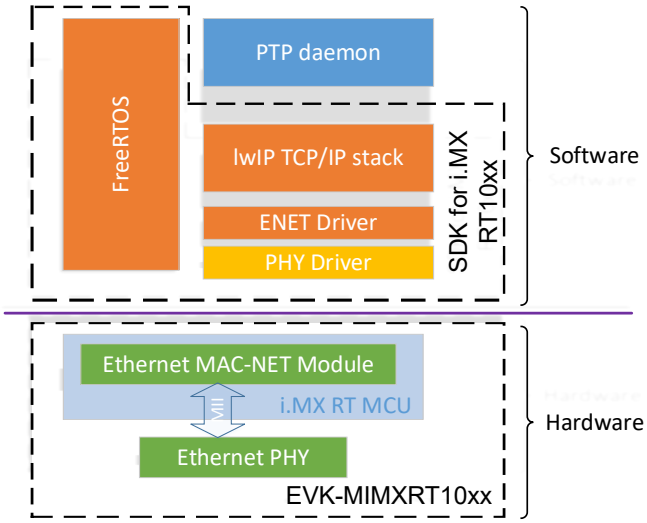

Figure 8. IEEE 1588 solution for i.MX RT10xx MCU

4.1 Hardware components

The i.MX RT10xx EVK board is the platform designed to showcase the most common features of the i.MX RT10xx processor. The i.MX RT10xx EVK board is an entry-level development board which helps you to quickly become familiar with the processor and expedites you to implement your own designs. The main features of the i.MX RT10xx EVK board include:

- i.MX RT10xx MCU
- 32MB@166 MHz SDRAM
- 512 Mbit Hyper Flash (only available for i.MX RT1050), 64 Mbit Quad SPI flash, and TF Card slot
- 10/100 Mbit/s Ethernet Connector with KSZ8081RNB PHY
- USB 2.0 OTG/Host Connectors
- 3.5 mm Audio Stereo Headphone Jack, Microphone, and Speaker out connectors
- Display connector and CMOS Sensor interface (unavailable for i.MX RT1020)
- CAN Bus connector, OpenSDA with DAP-Link, and Arduino interface
- 5 V DC-Jack for power supply

i.MX RT1050 is the first crossover processor in the industry, which is a new processor family featuring NXP's advanced implementation of the Arm Cortex-M7 core. It is designed to support the next-generation IoT applications with a high level of integration and security balanced with MCU-level usability. It operates at speeds of up to 600 MHz to provide high CPU performance with the best real-time functionality. i.MX RT 1050 provides various memory interfaces, including SDRAM, raw NAND flash, NOR flash, SD/eMMC, quad SPI, HyperBus, and a wide range of other interfaces to connect peripherals, such as WLAN, Bluetooth, GPS, display, and

camera sensors. As the other i.MX processors, i.MX RT1050 also integrates rich audio and video features including LCD display, basic 2D graphics, camera interface, and SPDIF and I^2S audio interfaces.

i.MX RT1020 provides a high-performance feature set in low-cost LQFP packages, further simplifying your board design and layout. This processor removes the multimedia component and reduces the on-chip SRAM to 256 KB for low-cost applications. i.MX RT1020 runs at 500 MHz.

For more information, refer to the corresponding reference manual on www.nxp.com.

4.2 Software components

The IEEE 1588 software implementation includes the MCUXpresso SDK IDE for the i.MX RT10xx EVB board and PTP daemon. The MCUXpresso SDK IDE is a software framework for developing applications on NXP MCUs including peripheral drivers, middleware, and real-time operating system.

4.2.1 FreeRTOS

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which you can build embedded applications that meet strict real-time requirements. FreeRTOS provides methods for multiple tasks, mutexes, semaphores, and software timers. A tickless mode is provided for low-power applications. The thread priorities used by the scheduler decide which thread should be executing. The version of the FreeRTOS provided by the MCUXpresso SDK IDE for the i.MX RT10xx EVB board is 10.0.1. The FreeRTOS package integrated into the MCUXpresso IDE has these features:

- Removed the files not related to the SDK IDE, such as extensions to the FreeRTOS (CLI, `FAT_SL`, and UDP) and folders, such as the demo and nested folders.
- Added the `SystemCoreClock` global variable to the FreeRTOS *port.c* and *FreeRTOSConfig.h* files.
- Enabled tickles mode. For more information, see the www.nxp.com/freertos.
- Enabled KDS Task Aware Debugger, Apply FreeRTOS patch to enable the `configRECORD_STACK_HIGH_ADDRESS` macro.
- Enable `-flto` optimization in GCC by adding `__attribute__((used))` for `vTaskSwitchContext`.

For detailed information about the FreeRTOS and its distribution, see www.freertos.org.

4.2.2 lwIP TCP/IP stack

lwIP is a light-weight implementation of the TCP/IP protocol suite that is freely available in the C source code and can be downloaded from the development webpage. It is completely modular and small enough to reduce the RAM usage for use in small embedded systems. The core stack is an IP implementation, on top of which you can choose to add TCP, UDP, DHCP, and many other protocols according to your needs and the memory available in the designed system. For more information about lwIP, see www.nongnu.org/lwip.

The MCUXpresso SDK IDE for EVK-MIMXRT1050 integrates the lwIP TCP/IP stack, which runs on top of the MCUXpresso SDK IDE Ethernet driver with the i.MX RT10xx EVB board. The lwIP package version in the SDK for the i.MX RT10xx EVB board is 2.15.x. For more information, see the *lwIP TCP/IP Stack and MCUXpresso SDK Integration User's Guide* (document MCUXSDKLWIPUG).

4.2.3 PTP daemon

PTP provides precise time coordination of Ethernet LAN-connected computers, which is designed primarily for instrumentation and control systems. PTP daemon (PTPd) is an open-source implementation of PTP version 2, as defined by IEEE Std 1588-2008.

PTPd coordinates the clocks of a group of LAN-connected computers with each other. It can achieve microsecond-level coordination even on the limited platform. PTPd is available in the C source code and can be ported on the FreeRTOS for embedded systems. Most of the system-related code is in the *<install_dir>/src/*

dep folder. The PTPd package version now used in this demo is version 2.3.1. It is available at github.com/ptpd/ptpd/releases.

5 Detailed description of the IEEE1588 demo software

This demo application now is only compiled and tested with the MCUXpresso IDE v11.9.0. The SDK version is 2.15.x and the PTPd version is 2.3.1. The i.MX RT10xx ENET supports IEEE 1588 with a hardware timestamp. To enable the hardware timestamp feature and run the demo on the i.MX RT10xx EVK board, the original lwIP TCP/IP port-related code must be updated. The update for the ENET driver is needed to test the clock offset converging and to get the right timestamp of 1588 related frames.

5.1 i.MX RT SDK ENET driver update

The ENET 1588 timer has four channels that support input capture and output compare using the 1588 counter. To monitor the synchronicity between the master and slave clocks while the demo is running, the ENET output compare feature must be enabled to generate a Pulse-Per-Second (PPS) signal while the free-running counter value matches the output compare reference value. If the master and slave clocks are synchronized properly and the output compare reference values of the master and slave are set the same, the 1588 timer output signals of the master and slave synchronization can be observed using an oscilloscope.

The code changes to enable the output compare in *fsl_enet.h* include the additional two members in the `struct _enet_handle` type.

The code in bold must be added or modified.

```
struct _enet_handle
{
   ……
#ifdef ENET_ENHANCEDBUFFERDESCRIPTOR_MODE
   enet_ptp_time_data_ring_t txPtpTsDataRing;
   enet_ptp_timer_channel_t mPtpTmrChannel; /*!< PTP 1588 timer channel. */
   uint32_t ptpNextCounter;                 /*!< PTP 1588 next output compare counter
 value */
#endif                                      /* ENET_ENHANCEDBUFFERDESCRIPTOR_MODE */
};
```

To enable the output compare feature in the IEEE 1588 timer, the code in bold must be added or modified in the `ENET_Ptp1588Configure` and `ENET_TimeStampIRQHandler` functions.

```
void ENET_Ptp1588Configure(ENET_Type *base, enet_handle_t *handle, enet_ptp_config_t
 *ptpConfig)
{
    ……
    handle->msTimerSecond = 0;
    handle->mPtpTmrChannel = ptpConfig->channel;
    ……
}
void ENET_TimeStampIRQHandler (ENET_Type *base, enet_handle_t *handle)
{
    ……
    if (kENET_TsAvailInterrupt & base->EIR)
    {
       ……
    }
    else if (base->CHANNEL[handle->mPtpTmrChannel].TCSR & ENET_TCSR_TF_MASK)
    {
        ENET_Ptp1588SetChannelCmpValue(base, handle->mPtpTmrChannel, handle-
>ptpNextCounter);
        do {
```

```
            ENET_Ptp1588ClearChannelStatus(base, handle->mPtpTmrChannel);
        } while (true == ENET_Ptp1588GetChannelStatus(base, handle->mPtpTmrChannel));
    }
    ......
}
```

Because the newest SDK removes the timestamp store function on low level driver, the transmit timestamp must be got from the Tx Buffer Descriptor directly. In 1588 protocol, only the timestamps of event frames are needed, so add some code to the `ENET_SendFrame()` function and use a global variable to store the timestamp of the sent frame. For the code changes, see AN12149SW.

5.2 lwIP TCP/IP porting update

This section describes the modifications of the lwIP porting code to support the PTP demo. This involves the *lwipopts.h*, *ethernetif.h*, and *ethernetif.c* files in the *<sdk_install_dir>/middleware/lwip/port* folder.

The PTP daemon demo uses the `SO_REUSEADDR` option for the socket, Domain Name System (DNS) protocol, and Internet Group Management Protocol (IGMP) protocol. It does not use the Dynamic Host Configuration Protocol (DHCP) with a static IP address.

Due to the newest SDK remove the timestamp store function, enable the `pbuf` custom data structure to store the timestamp of the received frames.

The following macros in *lwipopts.h* must be defined to the corresponding values:

```
/* SO_REUSE ==1: Enable SO_REUSEADDR option */
#define SO_REUSE  1

#define LWIP_PBUF_CUSTOM_DATA \
                u64_t t_sec; \
                u32_t t_nsec;

#ifndef LWIP_DHCP
#define LWIP_DHCP  0
#endif

/* ---------- DNS options ---------- */
#ifndef LWIP_DNS
#define LWIP_DNS  1
#endif

/* ---------- IGMP options ---------- */
/* LWIP_IGMP==1: Turn on IGMP module. */
#ifndef LWIP_IGMP
#define LWIP_IGMP  1
#endif
```

The default lwIP package in the SDK release does not support PTP. The ENET initializing function for lwIP does not involve any 1588 timer functions. The code update for *ethernetif.h* and *enet_ethernetif_kinetis.c* mainly covers the ENET 1588 timer routines, such as initializing the 1588 timer, enabling the timer channel output compare function for the test, setting/getting time, adjusting the 1588 timer frequency, and getting the timestamp of the transmit frames. All the code related to PTP is enclosed by the `#if LWIP_PTP` and `#endif` pair.

This code snippet shall be added to *ethernetif.h* to declare the routines of the 1588 timer:

```
#if LWIP_PTP
#include "lwip_ptp.h"

#define ENET_NANOSECOND_ONE_SECOND 1000000000U
#define PTP_AT_INC                          (ENET_NANOSECOND_ONE_SECOND/PTP_CLOCK_FRE_RT)

void ethernet_ptptime_settime(enet_ptp_time_t *timestamp);
```

```
void ethernet_ptptime_gettime(enet_ptp_time_t *timestamp);
void ethernet_ptptime_adjfreq(int32_t ppb);
err_t enet_get_rxframe_time(enet_ptp_time_data_t *ptpTimeData);
err_t enet_get_txframe_time(enet_ptp_time_data_t *ptpTimeData);
#endif
```

The above functions are implemented in the *enet_ethernetif_kinetis.c* file. The ENET 1588 timer-initializing function `ethernet_ptptime_init ()` is implemented and called before returning from the `enet_init()` function. The `ethernet_ptptime_enablepps()` function is implemented to enable the timer channel output compare function for test purposes and called in the `ethernet_ptptime_init ()` function according to the passed parameter.

This is the code of the `ethernet_ptptime_enablepps()` function:

```
static void ethernet_ptptime_enablepps(struct ethernetif *ethernetif,
                                       enet_ptp_timer_channel_t tmr_ch)
{
    uint32_t next_counter = 0;
    uint32_t tmp_val = 0;

    /* clear capture or output compare interrupt status if have. */
    ENET_Ptp1588ClearChannelStatus(ethernetif->base, tmr_ch);

    /* It is recommended to double check the TMODE field in the
     * TCSR register to be cleared before the first compare counter
     * is written into TCCR register. Just add a double check. */
    tmp_val = ethernetif->base->CHANNEL[tmr_ch].TCSR;
    do {
        tmp_val &= ~(ENET_TCSR_TMODE_MASK);
        ethernetif->base->CHANNEL[tmr_ch].TCSR = tmp_val;
        tmp_val = ethernetif->base->CHANNEL[tmr_ch].TCSR;
    } while (tmp_val & ENET_TCSR_TMODE_MASK);

    tmp_val = (ENET_NANOSECOND_ONE_SECOND >> 1);

    ENET_Ptp1588SetChannelCmpValue(ethernetif->base, tmr_ch, tmp_val);

    /* Calculate the second the compare event timestamp */
    next_counter = tmp_val;

    /* Compare channel setting. */
    ENET_Ptp1588ClearChannelStatus(ethernetif->base, tmr_ch);

    ENET_Ptp1588SetChannelOutputPulseWidth(ethernetif->base, tmr_ch, false, 4, true);

    /* Write the second compare event timestamp and calculate
     * the third timestamp. Refer the TCCR register detail in the spec.*/
    ENET_Ptp1588SetChannelCmpValue(ethernetif->base, tmr_ch, next_counter);
    /* Update next counter */
    ethernetif->handle.ptpNextCounter = next_counter;
}
```

This code is the ENET 1588 timer initializing function `ethernet_ptptime_init()` and its related memories:

```
static struct ethernetif * ptp_ethernetif  = NULL;

/* Buffers for store receive and transmit timestamp. */
enet_ptp_time_data_t g_rxPtpTsBuff[ENET_RXBD_NUM];
enet_ptp_time_data_t g_txPtpTsBuff[ENET_TXBD_NUM]

static void ethernet_ptptime_init(struct ethernetif *ethernetif, uint32_t ptp_clk_freq,
                                  bool pps_en, enet_ptp_timer_channel_t tmr_ch)
{
  enet_ptp_config_t ptp_cfg;
```

```
  assert(ethernetif);
  ptp_ethernetif = ethernetif;

  /* Config 1588 */
  memset(&ptp_cfg, 0, sizeof(enet_ptp_config_t));
  ptp_cfg.channel = tmr_ch;  ptp_cfg.ptp1588ClockSrc_Hz = ptp_clk_freq;
  ENET_Ptp1588Configure(ptp_ethernetif->base, &ptp_ethernetif->handle, &ptp_cfg);

  if (true == pps_en)
  {
    ethernet_ptptime_enablepps(ptp_ethernetif, tmr_ch);
  }
  else
  {
    ENET_Ptp1588SetChannelMode(ptp_ethernetif->base, tmr_ch, kENET_PtpChannelDisable,
 false);
  }
}
```

The syntax in bold is used to call `ethernet_ptptime_init ()` function in the `enet_init()` function:

```
static void enet_init(struct netif *netif, struct ethernetif *ethernetif,
                      const ethernetif_config_t *ethernetifConfig)
{
    ……
#if LWIP_PTP
    /* It's time to initialize the IE1588 function of ethernet */
    ethernet_ptptime_init(ethernetif, PTP_CLOCK_FRE_RT, PTP_TEST_APP_ENABLE,
                          PTP_TEST_APP_CHANNEL);
#endif
    ENET_ActiveRead(ethernetif->base);
}
```

The `ethernet_ptptime_adjfreq()` function adjusts the 1588 timer frequency by setting the non-zero correction counter wrap-around value in the `ENET_ATCOR` register to define the number of timer clock cycles to correct the 1588 timer's time. The correction increment value is set in the `INC_CORR` field in the `ENET_ATINC` register. The value of `INC_CORR` is bigger than the value in the INC field to speed up the 1588 timer. The value of `INC_CORR` is smaller than the value in the INC field to slow down the 1588 timer.

This is the code of the `ethernet_ptptime_adjfreq()` function:

```
void ethernet_ptptime_adjfreq(int32_t incps)
{
    int32_t neg_adj = 0;
    uint32_t corr_inc, corr_period;

    assert(ptp_ethernetif);

    /*
     * incps means the increment rate (nanseconds per second)by which to
     * slow down or speed up the slave timer.
     * Positive ppb need to speed up and negative value need to slow down.
    */

    if (0 == incps)
    {
        ptp_ethernetif->base->ATCOR &= ~ENET_ATCOR_COR_MASK; /* Reset PTP
 frequency */
        return;
    }
```

```
    if (incps < 0)
    {
        incps = - incps;
        neg_adj = 1;
    }

    corr_period = (uint32_t)PTP_CLOCK_FRE_RT / incps;

    /* neg_adj = 1, slow down timer, neg_adj = 0, speed up timer */
    corr_inc = (neg_adj) ? (PTP_AT_INC - 1) : (PTP_AT_INC + 1);

    ENET_Ptp1588AdjustTimer(ptp_ethernetif->base, corr_inc, corr_period);
}
```

The `ethernet_ptptime_settime()`, `ethernet_ptptime_gettime()`, `enet_get_rxframe_time()`, and `enet_get_txframe_time()` are implemented simply to wrap `ENET_Ptp1588GetTimer()`, `ENET_Ptp1588SetTimer()`, `ENET_GetRxFrameTime()`, and `ENET_GetTxFrameTime()` functions in the *enet_ethernetif_kinetis.c* file.

5.3 PTPd porting on FreeRTOS

The default PTPd source code is for the FreeBSD, NetBSD, Mac operating system X, and Linux operation systems. To port the code to FreeRTOS with the lwIP and SDK drivers for the i.MX RT10xx EVB board, the operating system-related code, network-related code, and hardware timestamping code must be ported or added. The ported work covers the PTPd tasks under the FreeRTOS, system time routines, system services, software timer, interaction with network socket, and minor modification of the PTP protocol. The simple code modifications required by the MCUXPresso IDE during compiling and linking are not discussed in this document.

Even the code in the files in the *ptpd/src* folder is common to the PTPd application. Some files must be updated for the PTPd to work on the FreeRTOS. The original `main()` function in the *ptpd.c* file is changed to `ptpd_thread()`, which is created as a FreeRTOS task. The original command line parameters are removed to simplify the demo functions, except for the variable to denote the master or the slave. This variable's value is passed by the parameter while the FreeRTOS task is being created.

Another significant modification in the PTPd common code is in the *protocol.c* file. The default PTPd application runs as a real network node within a UNIX-style system. The device can receive the frames of event messages sent by itself, because they are sent using UDP/IP multicast messages. The `Follow_Up` or `Pdelay_Resp_Follow_Up` messages are sent after the device receives corresponding `Sync or Pdelay_Resp` event messages sent by itself in the original protocol source code. Because this demo runs with a point-to-point connection, the device does not receive the event message sent by itself. The code must be modified to send these two follow-up messages as soon as the corresponding event message is sent. As a result of this change, the `netSelect()` function must be called with a specific timeout value to replace the original NULL (no timeout) that blocks the `select()` function waiting for a file descriptor indefinitely.

The files in the *ptpd/src/dep* folder are port-specific source code files and depend on the operating system, TCP/IP stack, and hardware platform. The main changes involve the `net.c`, `startup.c`, `sys.c`, and `timer.c` files. Their names are suffixed by `_mcu` to distinguish them from the original files.

FreeRTOS provides software timer functionality if setting `configUSE_TIMERS` to 1 in the `FreeRTOSConfig.h`. The modification for `eventtimer_itimer.c` includes the following two functions:

```
void startEventTimers(void)
{
     TimerHandle_t xptpTimer;

     xptpTimer = xTimerCreate("ptp_timer", pdMS_TO_TICKS(MS_TIMER_INTERVAL), pdTRUE,
 NULL, timerSignalHandler);
```

```
     if(xptpTimer != NULL)
     {
            xTimerStart(xptpTimer, portMAX_DELAY);
     }
}

static void timerSignalHandler(TimerHandle_t xTimer)
{
     (void)xTimer;

     elapsed++;
     /* be sure to NOT call DBG in asynchronous handlers! */
}
```

The `sys_mcu.c` file has some time-related routines to provide interfaces to the low-level hardware timer that is synchronized in the demo. Most of these routines call the functions described in Section 5.2.

The `adjFreq()` function code is as follows:

```
Boolean adjFreq(Integer32 adj)
{
    if (adj > ADJ_FREQ_MAX)
        adj = ADJ_FREQ_MAX;
    else if (adj < -ADJ_FREQ_MAX)
        adj = -ADJ_FREQ_MAX;
    ethernet_ptptime_adjfreq(adj);
    return TRUE;
}
```

There are two sleep functions to put the current thread into a dormant state that are implemented using the FreeRTOS `vTaskDelay()` function. The remaining changes just remove the code for the information output and/or the log file.

```
Boolean nanoSleep(TimeInternal * t)
{
    TickType_t time;

    time = pdMS_TO_TICKS(t->seconds * 1000 + t->nanoseconds / 1000000);
    vTaskDelay(time);
    return TRUE;

}

void milliSleep(int milli_seconds)
{
    TickType_t time;

    time = pdMS_TO_TICKS(milli_seconds);
    vTaskDelay(time);
}
```

The *startup_mcu.c* file removes all operating system-related signal functions and the command line parameter-parsing code. The `ptpd_init()` function is added to create a FreeRTOS operating system task for the PTPd application.

The timestamp of the transmit frame in the original code is provided by the operating system and the timestamp of the received frame can be extracted from the received data after calling the `recvmsg()` function which is supported by a Linux-style operating system. This demo uses the hardware timestamping feature in the ENET 1588 timer. These codes must be ported to the FreeRTOS and the ENET 1588 timer on the i.MX RT10xx MCUs. The code in the *net_mcu.c* file provides the ported functions.

The `getInterfaceAddress()` function directly returns the default network interface used in lwIP and implemented as follows:

```
static int getInterfaceAddress(char* ifaceName, int family, struct sockaddr* addr) {

    int ret;
    (void)family;
    struct ifaddrs *ifaddr, *ifa;

    struct netif * iface;
    iface = netGetDefaultNetif();

    memcpy(&((struct sockaddr_in*)addr)->sin_addr, &iface->ip_addr, sizeof (iface-
>ip_addr));

    ret = 1;

    return ret;
}
```

Because the ported code does not use the `recvmsg()` function to enable timestamps, the `netInitTimestamping()` function is not called in the `netInit()` function nor implemented as a dump function just returning TRUE.

The ENET driver on the i.MX RT10xx MCU now does not provide the APIs to directly query the timestamp of the event message's frame. As mentioned before, a global variable is used to store the Tx BD to get the timestamp of the transmitted frame. The timestamp of the received frame is stored in the `pbuf` which allocated by LwIP stack, in socket receive function, `ptpd` can get the timestamp directly. To get the timestamp, the ENET PTP message data and the timestamp data defined by the `enet_ptp_time_data_t` type shall be packed from the buffer that contains the received or sent event messages. The `netPackPtpData()` function is added for this task and listed explicitly in this code.

```
static void netPackPtpData(Octet * buf, enet_ptp_time_data_t *pptpTimeData)
{
    pptpTimeData->messageType = (*(Enumeration4 *) (buf + 0)) & 0x0F;
    pptpTimeData->sequenceId = flip16(*(UInteger16 *) (buf + 30));
    pptpTimeData->version = (*(UInteger4 *) (buf + 1)) & 0x0F;
    memcpy(pptpTimeData->sourcePortId, (buf + 20), 10);
}
```

The `recv()` socket function replaces the `recvmsg()` function in the `netRecvGeneral()` function to read the general message's frame. This is the code of the `netRecvGeneral()` function implementation:

```
ssize_t netRecvGeneral(Octet * buf, TimeInternal * time, NetPath * netPath)
{
    ssize_t ret;

    ret = recv(netPath->generalSock, buf, PACKET_SIZE, MSG_DONTWAIT);
    if (ret <= 0) {
        if (errno == EAGAIN || errno == EINTR)
            return 0;
        return ret;
    }

    return ret;
}
```

The `netRecvEvent()` function reads the event message's frame and returns its timestamp provided by the `recv` function of the LwIP. This is the code of its implementation:

```
ssize_t netRecvEvent(Octet * buf, TimeInternal * time, NetPath * netPath)
```

```
{
    ssize_t ret = 0;
    enet_ptp_time_data_t ptpTimeData;
  struct lwip_sock *sock;

    getTime(time); /* Read the current time used for timestame in case of reading from
 driver failed. */
    ret = recv(netPath->eventSock, buf, PACKET_SIZE, MSG_PEEK);
    if (ret <= 0) {
        if (errno == EAGAIN || errno == EINTR)
            return 0;

        return ret;
    }

    /* get time stamp of packet */
    if (!time) {
            ERROR("null receive time stamp argument\n");
            return 0;
    }

    netPackPtpData(buf, &ptpTimeData);

    sock = lwip_socket_dbg_get_socket(netPath->eventSock);

    time->nanoseconds = sock->lastdata.netbuf->p->t_nsec;
    time->seconds = (Integer32)sock->lastdata.netbuf->p->t_sec;

    ret = recv(netPath->eventSock, buf, PACKET_SIZE, MSG_DONTWAIT);
    if (ret <= 0) {
        if (errno == EAGAIN || errno == EINTR)
            return 0;

        return ret;
    }

    return ret;
}
```

Both the `netSentEvent()` and `netSentPeerEvent()` functions send the frame of corresponding event message and return the frame's timestamp. The default implementation doesn't support hardware timestamping. To enable hardware timestamping, the `netSentEvent()` and `netSentPeerEvent()` functions add another input parameter of pointer to the buffer that contains the returned timestamp.

This code snippet shows the code added into the `netSentEvent()` and `netSentPeerEvent()` functions in bold:

```
ssize_t netSendEvent(Octet * buf, UInteger16 length, TimeInternal * time, NetPath *
 netPath, Integer32 alt_dst)
{
    ……
    enet_ptp_time_data_t ptpTimeData;
    ……
    getTime(tim); /* Read the current time used for timestame in case of reading from
 driver failed. */

    netPackPtpData(buf, &ptpTimeData);

    if(!enet_get_txframe_time(&ptpTimeData)){
       /* return the timestamp gotten from driver */
       tim->nanoseconds = ptpTimeData.timeStamp.nanosecond;
       tim->seconds = (Integer32)ptpTimeData.timeStamp.second;
    }
    return ret;
```

```
}
```

The newest 2.3.1 PTPd source code still exists much code which adapts to the desktop operating system, so this application note also modifies many other files, most of which are removing the code that do not adapt to the RTOS. For all these changes, see AN12149SW.

5.4 FreeRTOS tasks and board configuration

There are three task threads created using the FreeRTOS in the demo application.

- `stack_init` task - Created in the main function. This task initializes the lwIP TCP/IP stack and the static IP address setting, netmask configuration, gateway address configuration, MAC address configuration, and Ethernet hardware initialization. Then it starts the PTPd task. Lastly, the task deletes itself by calling the `vTaskDelete()` function after initialing lwIP and PTPd tasks.
- `tcpip_thread` task - Created by the `stack_init` task during TCP/IP initialization. This task runs the main lwIP task to access lwIP core functions.
- `ptpd_thread` task - Created by the `stack_init` task. This task runs the PTPd application. The parameter passed while this task is being created denotes either the master or the slave.

The demo enables the one-channel output compare function of the 1588 timer. Its output signal is asserted according to the configuration while the output compare event happens.

Channel 3 of the 1588 timer is used to generate an output compare event in this demo for the i.MX RT1050 MCUs. The output signal is routed to the `GPIO_AD_B1_02` pin. The following syntax configures `GPIO_AD_B1_02` as `ENET_1588_EVNT2_OUT` (output signal of channel 3) in the *pin_mux.c* file:

```
/* GPIO_AD_B1_02 is configured as 1588_EVENT2_OUT */
IOMUXC_SetPinMux(IOMUXC_GPIO_AD_B1_02_ENET_1588_EVENT2_OUT, 0U);

/* GPIO_AD_B0_12 PAD functional properties */
IOMUXC_SetPinConfig(IOMUXC_GPIO_AD_B1_02_ENET_1588_EVENT2_OUT, 0x10B0u);
```

Channel 2 of the 1588 timer is used to generate an output compare event in this demo for the i.MX RT1020 MCU. The output signal is routed to the `GPIO_SD_B1_02` pin. This syntax configures `GPIO_SD_B1_02` as `ENET_1588_EVNT1_OUT` (output signal of channel 2) in the *pin_mux.c* file:

```
/* GPIO_SD_B1_02 is configured as 1588_EVENT1_OUT */
IOMUXC_SetPinMux(IOMUXC_GPIO_SD_B1_02_ENET_1588_EVENT1_OUT, 0U);

/* GPIO_SD_B0_12 PAD functional properties */
IOMUXC_SetPinConfig(IOMUXC_GPIO_SD_B1_02_ENET_1588_EVENT1_OUT, 0x10B0u);
```

The 1588 timer is clocked from *ref_enetpll2* (generated by the Ethernet PLL) which must be enabled. The Ethernet PLL is initialized as follows:

```
void BOARD_InitModuleClock(void)
{
    const clock_enet_pll_config_t config = {true, true, 1, 0};
    CLOCK_InitEnetPll(&config);
}
```

The other board-specific initialization codes are the same as the `enet_txrx_ptp1588` example in the *<sdk_install_dir/boards/evkmimxrt1050/driver_examples/enet/txrx_ptp1588_transfer* folder.

Because the MCUXpresso IDE exists the issue that when importing the PTPd project that this application note is related, it also compiles the *NO_SYS_SampleCode.c* and *ZeroCopyRx.c* files in the */lwip/doc* folder. Manually exclude them from build.

6 Running the IEEE1588 demo

The section describes how to set up the 1588 demo using the i.MX RT10xx EVK board and the demo software described in the above sections.

6.1 Hardware setup

To set up the hardware for the test, two i.MX RT10xx EVK boards must be used and connected to each other. The demo has a point-to-point configuration where two boards are connected directly using the crossover Ethernet cable. This demo uses a simple type of connection often used to evaluate the system's accuracy and overall performance. Figure 9 shows the point-to-point configuration using two i.MX RT10xx EVK boards. No specific jumper settings are needed for the test.

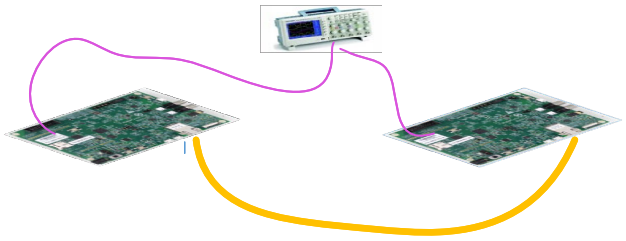

Figure 9. Back-to-back configuration of the demo

For detailed information on how to use the i.MX RT10xx EVK board and set its jumpers, see the *MIMXRT10xx EVK Board Hardware User's Guide*.

6.2 Clock synchronicity measuring

This demo can generate a pulse-per-second (PPS) signal to measure the synchronicity of the clocks between master and slave. As for the i.MX RT1050MCU, the PPS signal is generated directly from Channel 3 of the 1588 timer and configured to output through the `GPIO_AD_B1_02` pin. This GPIO signal is routed to the J22-7 pin of the Arduino interface. For the i.MX RT1020 MCU, the PPS signal is generated directly from Channel 2 of the 1588 timer and configured to output through the `GPIO_SD_B1_02` pin. This GPIO signal is routed to the J19-10 pin of the Arduino interface.

To measure and compare the PPS signals from two boards, attach two oscilloscope probes to the J22-7 pins on the i.MX RT1050 EVK boards respectively, and/or to the J19-10 pin on the i.MX RT1020 EVK board. The two boards are powered for a time interval in the test. The oscilloscope shows that the slave PPS signal moves closer and closer to the master PPS signal and the offset converges to vary between a range of four clock cycles of the 1588 timer.

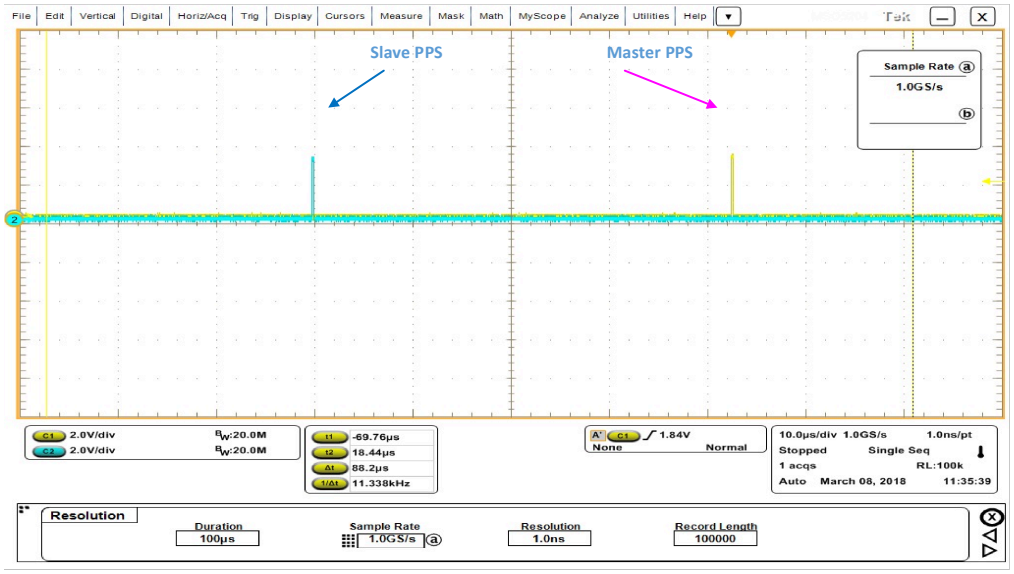

Figure 10. PTP startup - PPS distance between Master and Slave

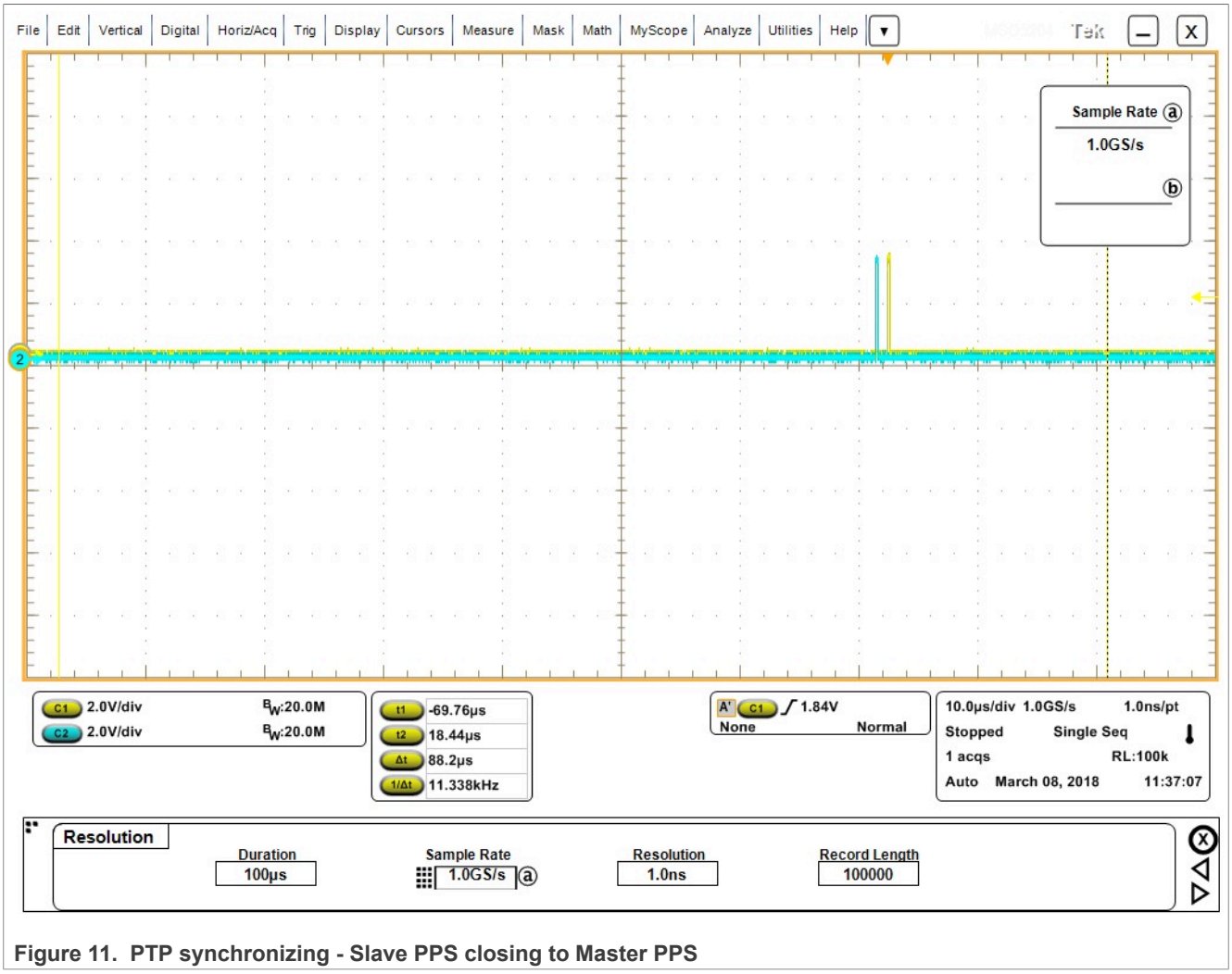

Figure 11. PTP synchronizing - Slave PPS closing to Master PPS

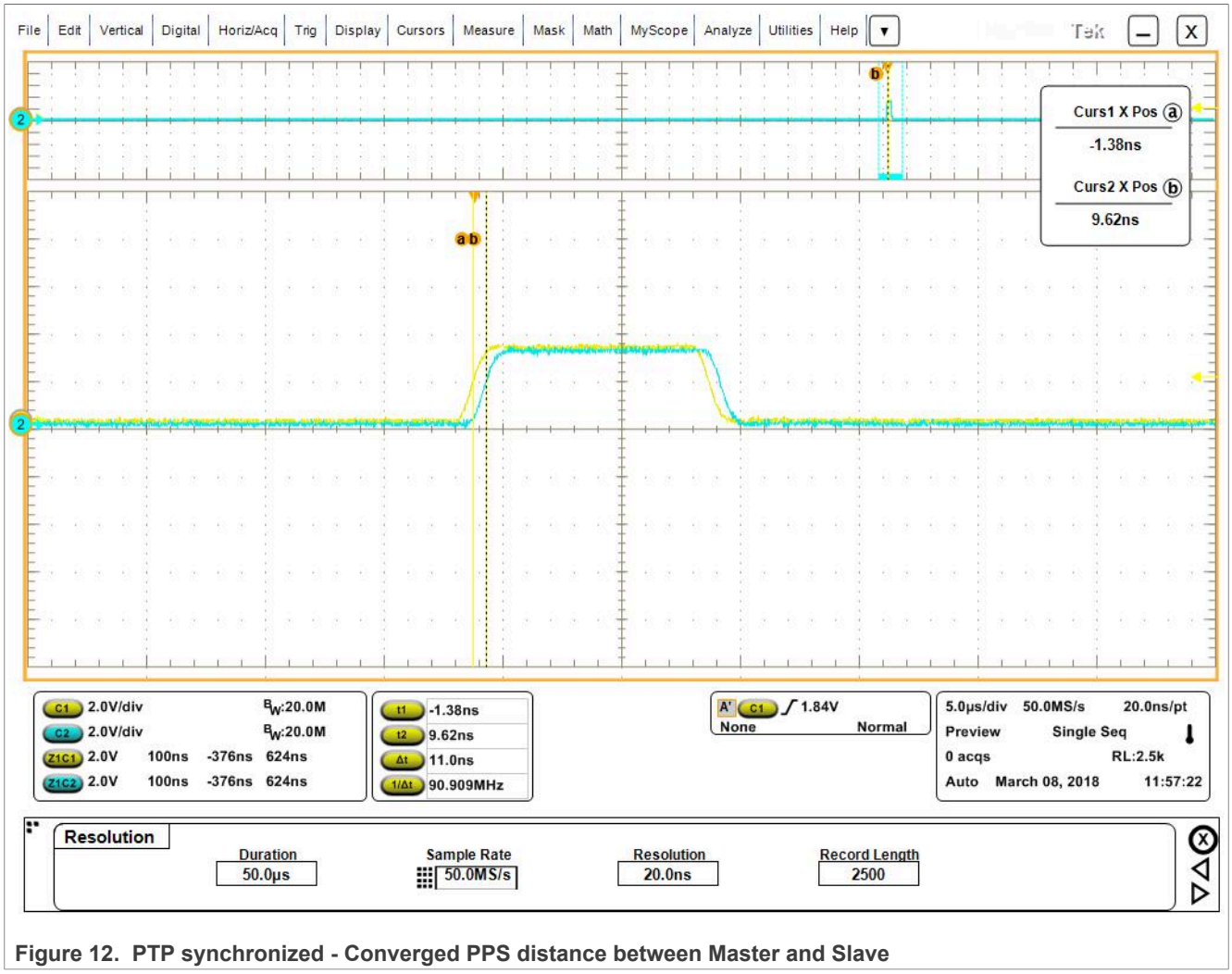

Figure 12. PTP synchronized - Converged PPS distance between Master and Slave

7 Conclusion

This application note describes the IEEE 1588 Precision Time Protocol demo application based on the open-source PTP daemon, FreeRTOS, lwIP TCP/IP stack, SDK for i.MX RT10xx, and the i.MX RT10xx Evaluation Kit (EVK-MIMXRT1050) board. This demo can be easily ported to other processors from the i.MX RT series with the FreeRTOS, lwIP, and TCP/IP stack support.

The demo system is targeted for applications that require precise clock synchronization between devices with accuracy in the sub-microsecond range.

Because the PTPd project has been out of maintenance for about 10 years and its original design target is used in a desktop operating system, this application note just implements its fundamental synchronization function. For further functions related to 1588, see the open source genAVB/TSN project developed by NXP.

8 Acronyms and abbreviations

Table 1. Acronyms

Acronyms	Meaning
API	Application Program Interface
BMC	Best Master Clock
DHCP	Dynamic Host Control Protocol
DNS	Domain Name System
ENET	10/100-Mbps Ethernet MAC
GPIO	General Port Input Output
ICMP	Internet Control Message Protocol
IGMP	Internet Group Management Protocol
MAC	Media Access Control
PPS	Pulse-per-second
PTP	Precision Time Protocol
PTPd	PTP Daemon
RTOS	Real Time Operation System
SDK	Software Development Kits
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

9 Note about the source code in the document

The example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2025 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

10 Revision history

Table 2 summarizes the revisions to this document.

Table 2. Revision history

Document ID	Release date	Description
AN12149 v2.0	31 March 2025	• Updated the code to SDK2.4.x • Added support for RT1050, RT1060, and RT1020
AN12149 v1.0	25 September 2018	Initial public release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents