```
[7/16, 10:26] SP Nonunaut: #include <stdio.h>
#include "xparameters.h"
#include "netif/xadapter.h"
#include "platform.h"
#include "platform_config.h"
#include "xil_printf.h"
#include "sleep.h"
#include "xil_cache.h"

// lwIP Includes
#include "lwip/init.h"
#include "lwip/netif.h"
#include "lwip/inet.h"
#include "lwip/tcp.h"
#include "lwip/udp.h"
#include "lwip/priv/tcp_priv.h"
#include "lwipopts.h"

#if LWIP_DHCP
#include "lwip/dhcp.h"
extern volatile int dhcp_timoutcntr;
#endif

// PTP Includes
#include "ptp/ptpd.h"

// Hardware Driver Includes for Timer and Interrupts
#include "xintc.h"      // AXI Interrupt Controller Header
#include "xtmrctr.h"    // AXI Timer Header

// --- Constant Definitions ---
#define DEFAULT_IP_ADDRESS  "192.168.1.10"
#define DEFAULT_IP_MASK     "255.255.255.0"
#define DEFAULT_GW_ADDRESS  "192.168.1.1"

// ** IMPORTANT: Update these IDs to match your Vivado Block Design **
#define INTC_DEVICE_ID      XPAR_INTC_0_DEVICE_ID
#define TMRCTR_DEVICE_ID    XPAR_TMRCTR_0_DEVICE_ID
#define TIMER_IRPT_INTR     XPAR_INTC_0_TMRCTR_0_VEC_ID

// PTP periodic tick rate (10 Hz = 100ms)
#define PTP_TICK_RATE_HZ    10
#define TIMER_RESET_VALUE   (XPAR_AXI_TIMER_0_CLOCK_FREQ_HZ /
PTP_TICK_RATE_HZ)

// --- Global Variables ---
extern volatile int TcpFastTmrFlag;
extern volatile int TcpSlowTmrFlag;
```

```
struct netif server_netif;
static XIntc interrupt_controller;
static XTmrCtr timer_controller;

// Flag set by the timer ISR to trigger PTP processing
volatile int ptp_timer_flag = 0;

// PTP Globals
ptp_clock_t ptp_clock;
ptpd_opts ptp_opts;
foreign_master_record_t foreign_records[PTPD_DEFAULT_MAX_FOREIGN_RECORDS];
sys_mbox_t ptp_alert_queue;

// --- Function Prototypes ---
static int setup_interrupt_system();
static void Timer_ISR_Handler(void *CallBackRef, u8 TmrCtrNumber);

// --- IP Address Helper Functions ---
static void print_ip(char *msg, ip_addr_t *ip)
{
    xil_printf("%s: %d.%d.%d.%d\r\n", msg,
            ip4_addr1(ip), ip4_addr2(ip), ip4_addr3(ip), ip4_addr4(ip));
}

static void print_ip_settings(ip_addr_t *ip, ip_addr_t *mask, ip_addr_t *gw)
{
    print_ip("Board IP", ip);
    print_ip("Netmask", mask);
    print_ip("Gateway", gw);
}

static void assign_default_ip(ip_addr_t *ip, ip_addr_t *mask, ip_addr_t *gw)
{
    inet_aton(DEFAULT_IP_ADDRESS, ip);
    inet_aton(DEFAULT_IP_MASK, mask);
    inet_aton(DEFAULT_GW_ADDRESS, gw);
}

// --- PTP Initialization ---
void ptpd_opts_init()
{
    xil_printf("Initializing ptpd options...\r\n");

    memset(&ptp_opts, 0, sizeof(ptpd_opts));
    ptp_opts.slave_only = 0;
    ptp_opts.sync_interval = 1;
    ptp_opts.announce_interval = 1;
    ptp_opts.clock_quality.clock_class = 248;
    ptp_opts.clock_quality.clock_accuracy = 0xFE;
    ptp_opts.clock_quality.offset_scaled_log_variance = 0xFFFF;
    ptp_opts.priority1 = 128;
    ptp_opts.priority2 = 128;

    if (ptp_startup(&ptp_clock, &ptp_opts, foreign_records) != 0) {
        xil_printf("PTP startup failed!\r\n");
    }
}

// --- Main Application ---
int main()
{
    struct netif *netif = &server_netif;

    unsigned char mac_ethernet_address[] = {
        0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };

    init_platform();

    xil_printf("\r\n----- PTP + lwIP UDP Server (Bare-Metal) -----\r\n");

    lwip_init();

    if (!xemac_add(netif, NULL, NULL, NULL, mac_ethernet_address,
                       PLATFORM_EMAC_BASEADDR)) {
        xil_printf("Error adding network interface\r\n");
        return -1;
    }
    netif_set_default(netif);

    // This enables interrupts globally, including for the timer
    platform_enable_interrupts();
    setup_interrupt_system(); // Setup timer interrupt

    netif_set_up(netif);

#if LWIP_DHCP
    dhcp_start(netif);
    dhcp_timoutcntr = 240;
    while ((netif->ip_addr.addr == 0) && (dhcp_timoutcntr > 0)) {
        xemacif_input(netif);
    }

    if (netif->ip_addr.addr == 0) {
        xil_printf("DHCP timeout! Assigning static IP.\r\n");
        assign_default_ip(&(netif->ip_addr), &(netif->netmask), &(netif->gw));
    }
#else
    assign_default_ip(&(netif->ip_addr), &(netif->netmask), &(netif->gw));
#endif

    print_ip_settings(&(netif->ip_addr), &(netif->netmask), &(netif->gw));

    // Create ptp_alert_queue
    ptp_alert_queue = sys_mbox_new();

    // Setup ptpd and register UDP handlers
    ptpd_opts_init();
    ptpd_net_init(&ptp_clock.net_path);

    xil_printf("PTP initialized. Starting main loop...\r\n");

    while (1) {
        // Handle lwIP's own timers (if TCP is used)
        if (TcpFastTmrFlag) {
            tcp_fasttmr();
            TcpFastTmrFlag = 0;
        }
        if (TcpSlowTmrFlag) {
            tcp_slowtmr();
            TcpSlowTmrFlag = 0;
        }

        // Poll for incoming network packets
        xemacif_input(netif);

        // Check if the periodic timer has fired
        if (ptp_timer_flag) {
            ptp_timer_flag = 0; // Reset the flag
            ptpd_periodic_handler(); // Run the PTP state machine
        }
    }

    cleanup_platform();
    return 0;
}

// --- Timer and Interrupt Setup Functions ---

void Timer_ISR_Handler(void *CallBackRef, u8 TmrCtrNumber)
{
    // Set the flag for the main loop to process
    ptp_timer_flag = 1;
}
```

```
static int setup_interrupt_system()
{
    int status;

    // Initialize the interrupt controller driver
    status = XIntc_Initialize(&interrupt_controller, INTC_DEVICE_ID);
    if (status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    // Initialize the timer driver
    status = XTmrCtr_Initialize(&timer_controller, TMRCTR_DEVICE_ID);
    if (status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    // Connect the timer ISR to the interrupt controller
    status = XIntc_Connect(&interrupt_controller, TIMER_IRPT_INTR,
                           (XInterruptHandler)XTmrCtr_InterruptHandler,
                           &timer_controller);
    if (status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    // Start the interrupt controller
    status = XIntc_Start(&interrupt_controller, XIN_REAL_MODE);
    if (status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    // Enable the timer interrupt in the interrupt controller
    XIntc_Enable(&interrupt_controller, TIMER_IRPT_INTR);

    // Set the timer handler that will be called from the driver's ISR
    XTmrCtr_SetHandler(&timer_controller, Timer_ISR_Handler, NULL);

    // Configure the timer for auto-reload (periodic) mode
    XTmrCtr_SetOptions(&timer_controller, 0, XTC_INT_MODE_OPTION |
XTC_AUTO_RELOAD_OPTION);

    // Set the timer reset value for a 10 Hz tick rate
    XTmrCtr_SetResetValue(&timer_controller, 0, TIMER_RESET_VALUE);

    // Start the timer
    XTmrCtr_Start(&timer_controller, 0);

    xil_printf("Periodic timer for PTP started successfully.\r\n");
    return XST_SUCCESS;
}
[7/16, 10:34] SP Nonunaut: #ifndef LWIP_HDR_APPS_PTPD_H
#define LWIP_HDR_APPS_PTPD_H

/* #define PTPD_DBGVV */
/* #define PTPD_DBGV */
/* #define PTPD_DBG */
/* #define PTPD_ERR */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>

#include "lwip/opt.h"
#include "lwip/api.h"
#include "lwip/inet.h"
#include "lwip/mem.h"
#include "lwip/udp.h"
#include "lwip/igmp.h"
#include "lwip/arch.h"

#include "ptpd_opts.h"
#include "ptpd_constants.h"
#include "ptpd_datatypes.h"

#ifdef __cplusplus
extern "C" {
#endif

/** Debug messages **/
#ifdef PTPD_DBGVV
#define PTPD_DBGV
#define PTPD_DBG
#define PTPD_ERR
#define DBGVV(...) printf("(V) " __VA_ARGS__)
#else
#define DBGVV(...)
#endif

#ifdef PTPD_DBGV
#define PTPD_DBG
#define PTPD_ERR
#define DBGV(...)  { TimeInternal tmpTime; getTime(&tmpTime); printf("(d %d.%09d) ", tmpTime.seconds, tmpTime.nanoseconds); printf(__VA_ARGS__); }
```

```
#else
#define DBGV(...)
#endif

#ifdef PTPD_DBG
#define PTPD_ERR
#define DBG(...)  { TimeInternal tmpTime; getTime(&tmpTime); printf("(D %d.%09d) ", tmpTime.seconds, tmpTime.nanoseconds); printf(__VA_ARGS__); }
#else
#define DBG(...)
#endif

#ifdef PTPD_ERR
#define ERROR(...)  { TimeInternal tmpTime; getTime(&tmpTime); printf("(E %d.%09d) ", tmpTime.seconds, tmpTime.nanoseconds); printf(__VA_ARGS__); }
#else
#define ERROR(...)
#endif

/** Endian corrections **/
#define flip16(x) htons(x)
#define flip32(x) htonl(x)

/** Flag manipulation **/
#define getFlag(flagField, mask) (((flagField) & (mask)) == (mask))
#define setFlag(flagField, mask) ((flagField) |= (mask))
#define clearFlag(flagField, mask) ((flagField) &= ~(mask))

/** Inline min/max **/
static __inline int32_t max(int32_t a, int32_t b) { return a > b ? a : b; }
static __inline int32_t min(int32_t a, int32_t b) { return a < b ? a : b; }

/** msg.c - Pack and unpack PTP messages **/
void msg_unpack_header(const octet_t* buf, msg_header_t* header);
void msg_unpack_announce(const octet_t* buf, msg_announce_t* announce);
void msg_unpack_sync(const octet_t* buf, msg_sync_t* sync);
void msg_unpack_followup(const octet_t* buf, msg_followup_t* follow);
void msg_unpack_delay_req(const octet_t* buf, msg_delay_req_t* delayreq);
void msg_unpack_delay_resp(const octet_t* buf, msg_delay_resp_t* resp);
void msg_unpack_pdelay_req(const octet_t* buf, msg_pdelay_req_t* pdelayreq);
void msg_unpack_pdelay_resp(const octet_t* buf, msg_pdelay_resp_t* presp);
void msg_unpack_pdelay_resp_followup(const octet_t* buf, msg_pdelay_resp_followup_t* prespfollow);
void msgUnpackManagement(const octet_t* buf, msg_management* manage);
void msgUnpackManagementPayload(const octet_t *buf, msg_management* manage);
void msg_pack_header(const ptp_clock_t* ptpClock, octet_t* buf);
void msg_pack_announce(const ptp_clock_t* ptpClock, octet_t* buf);
void msg_pack_sync(const ptp_clock_t* ptpClock, octet_t* buf, const timestamp_t*
originTimestamp);
void msg_pack_followup(const ptp_clock_t* ptpClock, octet_t* buf, const timestamp_t*
preciseOriginTimestamp);
void msg_pack_delay_req(const ptp_clock_t* ptpClock, octet_t* buf, const timestamp_t*
originTimestamp);
void msg_pack_relay_resp(const ptp_clock_t* ptpClock, octet_t* buf, const msg_header_t*
header, const timestamp_t* receiveTimestamp);
void msg_pack_pdelay_req(const ptp_clock_t* ptpClock, octet_t* buf, const timestamp_t*
originTimestamp);
void msg_pack_pdelay_resp(octet_t* buf, const msg_header_t* header, const timestamp_t*
requestReceiptTimestamp);
void msg_pack_pdelay_resp_followup(octet_t* buf, const msg_header_t* header, const
timestamp_t* responseOriginTimestamp);
int16_t msgPackManagement(const ptp_clock_t*,  octet_t*, const msg_management*);
int16_t msgPackManagementResponse(const ptp_clock_t*,  octet_t*, msg_header_t*, const
msg_management*);

/** servo.c - Clock servo **/
void servo_init_clock(ptp_clock_t* clock);
void servo_update_peer_delay(ptp_clock_t* clock, const time_interval_t* correction_field,
bool is_two_step);
void servo_update_delay(ptp_clock_t* clock, const time_interval_t*
delay_event_egress_timestamp, const time_interval_t* recv_timestamp, const
time_interval_t* correction_field);
void servo_update_offset(ptp_clock_t* clock, const time_interval_t*
sync_event_ingress_timestamp, const time_interval_t* precise_origin_timestamp, const
time_interval_t* correction_field);
void servo_update_clock(ptp_clock_t* clock);

/** startup.c - Runtime options **/
void ptpd_opts_init(void);
int16_t ptp_startup(ptp_clock_t* clock, ptpd_opts* opts, foreign_master_record_t* foreign);
void ptpdShutdown(ptp_clock_t* clock);

/** sys.c - System time API **/
void displayStats(const ptp_clock_t* ptpClock);
bool nanoSleep(const time_interval_t*);
void sys_get_clocktime(time_interval_t* time);
void sys_set_clocktime(const time_interval_t* time);
void ptpd_update_time(const time_interval_t* time);
bool ptpd_adj_frequency(int32_t adj);
uint32_t sys_get_rand(uint32_t rand_max);

/** timer.c - Timer handling **/
void ptp_init_timer(void);
void ptp_timer_stop(int32_t index);
void ptp_timer_start(int32_t index, uint32_t interval_ms);
```

```
bool ptp_timer_expired(int32_t index);

/** arith.c - Timing arithmetic **/
void ptp_time_scaled_nanoseconds_to_internal(const int64_t* scaledNanoseconds,
time_interval_t* internal);
void ptp_time_from_internal(const time_interval_t* internal, timestamp_t* external);
void ptp_to_internal_time(time_interval_t* internal, const timestamp_t* external);
void ptp_time_add(time_interval_t* r, const time_interval_t* x, const time_interval_t* y);
void ptp_sub_time(time_interval_t* r, const time_interval_t* x, const time_interval_t* y);
void ptp_time_halve(time_interval_t* r);
int32_t ptp_floor_log2(uint32_t n);

/** bmc.c - Best Master Clock Algorithm **/
uint8_t bmc(ptp_clock_t*);
void bmc_m1(ptp_clock_t* clock);
void bmc_p1(ptp_clock_t* clock);
void bmc_s1(ptp_clock_t* clock, const msg_header_t* header, const msg_announce_t*
announce);
void bcm_init_data(ptp_clock_t* clock);
bool bmc_is_same_port_identity(const port_identity_t* A, const port_identity_t* B);
void bmc_add_foreign(ptp_clock_t* clock, const msg_header_t* header, const
msg_announce_t* announce);

/** protocol.c - Protocol engine **/
void ptp_do_state(ptp_clock_t*);
void ptp_to_state(ptp_clock_t* clock, uint8_t state);

/** ptpd network and API **/
bool ptpd_net_init(net_path_t* net_path, ptp_clock_t* clock);
bool ptpd_shutdown(net_path_t* net_path);
int32_t ptpd_net_select(net_path_t* net_path, const time_interval_t* timeout);
void ptpd_empty_event_queue(net_path_t* net_path);
ssize_t ptpd_recv_event(net_path_t* net_path, octet_t* buf, time_interval_t* time);
ssize_t ptpd_recv_general(net_path_t* net_path, octet_t* buf, time_interval_t* time);
ssize_t ptpd_send_event(net_path_t* net_path, const octet_t* buf, int16_t length,
time_interval_t* time);
ssize_t ptpd_send_general(net_path_t* net_path, const octet_t* buf, int16_t length);
ssize_t ptpd_peer_send_event(net_path_t* net_path, const octet_t* buf, int16_t length,
time_interval_t* time);
ssize_t ptpd_peer_send_general(net_path_t* net_path, const octet_t* buf, int16_t length);

/** System time functions **/
void sys_get_clocktime(time_interval_t* time);
void sys_set_clocktime(const time_interval_t* time);
void ptpd_update_time(const time_interval_t* time);
bool ptpd_adj_frequency(int32_t adj);

/** PTPD alert **/
```

```
void ptpd_alert(void);

#ifdef __cplusplus
}
#endif

#endif /* LWIP_HDR_APPS_PTPD_H */
[7/16, 10:35] SP Nonunaut: #include <stdio.h>
#include "xparameters.h"
#include "netif/xadapter.h"
#include "platform.h"
#include "platform_config.h"
#include "xil_printf.h"
#include "sleep.h"
#include "xil_cache.h"

// lwIP Includes
#include "lwip/init.h"
#include "lwip/netif.h"
#include "lwip/inet.h"
#include "lwip/tcp.h"
#include "lwip/udp.h"
#include "lwip/priv/tcp_priv.h"
#include "lwipopts.h"

#if LWIP_DHCP
#include "lwip/dhcp.h"
extern volatile int dhcp_timoutcntr;
#endif

// PTP Includes
#include "ptp/ptpd.h"

// Hardware Driver Includes for Timer and Interrupts
#include "xintc.h"      // AXI Interrupt Controller Header
#include "xtmrctr.h"    // AXI Timer Header

// --- Constant Definitions ---
#define DEFAULT_IP_ADDRESS  "192.168.1.10"
#define DEFAULT_IP_MASK     "255.255.255.0"
#define DEFAULT_GW_ADDRESS  "192.168.1.1"

#define INTC_DEVICE_ID      XPAR_INTC_0_DEVICE_ID
#define TMRCTR_DEVICE_ID    XPAR_TMRCTR_0_DEVICE_ID
#define TIMER_IRPT_INTR     XPAR_INTC_0_TMRCTR_0_VEC_ID

#define PTP_TICK_RATE_HZ    10
```

```
#define TIMER_RESET_VALUE   (XPAR_AXI_TIMER_0_CLOCK_FREQ_HZ /
PTP_TICK_RATE_HZ)

#define PTPD_MBOX_SIZE      8 // Mailbox size for ptp_alert_queue

// --- Global Variables ---
extern volatile int TcpFastTmrFlag;
extern volatile int TcpSlowTmrFlag;

struct netif server_netif;
static XIntc interrupt_controller;
static XTmrCtr timer_controller;

volatile int ptp_timer_flag = 0;

// PTP Globals
ptp_clock_t ptp_clock;
ptpd_opts ptp_opts;
foreign_master_record_t foreign_records[PTPD_DEFAULT_MAX_FOREIGN_RECORDS];
sys_mbox_t ptp_alert_queue;

// --- Function Prototypes ---
static int setup_interrupt_system();
static void Timer_ISR_Handler(void *CallBackRef, u8 TmrCtrNumber);

// --- IP Address Helper Functions ---
static void print_ip(char *msg, ip_addr_t *ip)
{
    xil_printf("%s: %d.%d.%d.%d\r\n", msg,
            ip4_addr1(ip), ip4_addr2(ip), ip4_addr3(ip), ip4_addr4(ip));
}

static void print_ip_settings(ip_addr_t *ip, ip_addr_t *mask, ip_addr_t *gw)
{
    print_ip("Board IP", ip);
    print_ip("Netmask", mask);
    print_ip("Gateway", gw);
}

static void assign_default_ip(ip_addr_t *ip, ip_addr_t *mask, ip_addr_t *gw)
{
    inet_aton(DEFAULT_IP_ADDRESS, ip);
    inet_aton(DEFAULT_IP_MASK, mask);
    inet_aton(DEFAULT_GW_ADDRESS, gw);
}

// --- PTP Initialization ---
void init_ptpd_opts()
```

```
{
    xil_printf("Initializing ptpd options...\r\n");

    memset(&ptp_opts, 0, sizeof(ptpd_opts));
    ptp_opts.slave_only = 0;
    ptp_opts.sync_interval = 1;
    ptp_opts.announce_interval = 1;
    ptp_opts.clock_quality.clock_class = 248;
    ptp_opts.clock_quality.clock_accuracy = 0xFE;
    ptp_opts.clock_quality.offset_scaled_log_variance = 0xFFFF;
    ptp_opts.priority1 = 128;
    ptp_opts.priority2 = 128;

    if (ptp_startup(&ptp_clock, &ptp_opts, foreign_records) != 0) {
        xil_printf("PTP startup failed!\r\n");
    }
}

// --- Main Application ---
int main()
{
    struct netif *netif = &server_netif;

    unsigned char mac_ethernet_address[] = {
        0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };

    init_platform();

    xil_printf("\r\n----- PTP + lwIP UDP Server (Bare-Metal) -----\r\n");

    lwip_init();

    if (!xemac_add(netif, NULL, NULL, NULL, mac_ethernet_address,
                   PLATFORM_EMAC_BASEADDR)) {
        xil_printf("Error adding network interface\r\n");
        return -1;
    }
    netif_set_default(netif);

    platform_enable_interrupts();
    setup_interrupt_system();

    netif_set_up(netif);

#if LWIP_DHCP
    dhcp_start(netif);
    dhcp_timoutcntr = 240;
    while ((netif->ip_addr.addr == 0) && (dhcp_timoutcntr > 0)) {
        xemacif_input(netif);
    }

    if (netif->ip_addr.addr == 0) {
        xil_printf("DHCP timeout! Assigning static IP.\r\n");
        assign_default_ip(&(netif->ip_addr), &(netif->netmask), &(netif->gw));
    }
#else
    assign_default_ip(&(netif->ip_addr), &(netif->netmask), &(netif->gw));
#endif

    print_ip_settings(&(netif->ip_addr), &(netif->netmask), &(netif->gw));

    // Create ptp_alert_queue
    ptp_alert_queue = sys_mbox_new(PTPD_MBOX_SIZE);

    // Setup ptpd and register UDP handlers
    init_ptpd_opts();
    ptpd_net_init(&ptp_clock.net_path);

    xil_printf("PTP initialized. Starting main loop...\r\n");

    while (1) {
        if (TcpFastTmrFlag) {
            tcp_fasttmr();
            TcpFastTmrFlag = 0;
        }
        if (TcpSlowTmrFlag) {
            tcp_slowtmr();
            TcpSlowTmrFlag = 0;
        }

        xemacif_input(netif);

        if (ptp_timer_flag) {
            ptp_timer_flag = 0;
            ptp_protocol(&ptp_clock);  // This replaces ptpd_periodic_handler()
        }
    }

    cleanup_platform();
    return 0;
}

// --- Timer and Interrupt Setup ---
void Timer_ISR_Handler(void *CallBackRef, u8 TmrCtrNumber)
{
    ptp_timer_flag = 1;
```

```
}

static int setup_interrupt_system()
{
    int status;

    status = XIntc_Initialize(&interrupt_controller, INTC_DEVICE_ID);
    if (status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    status = XTmrCtr_Initialize(&timer_controller, TMRCTR_DEVICE_ID);
    if (status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    status = XIntc_Connect(&interrupt_controller, TIMER_IRPT_INTR,
                           (XInterruptHandler)XTmrCtr_InterruptHandler,
                           &timer_controller);
    if (status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    status = XIntc_Start(&interrupt_controller, XIN_REAL_MODE);
    if (status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    XIntc_Enable(&interrupt_controller, TIMER_IRPT_INTR);

    XTmrCtr_SetHandler(&timer_controller, Timer_ISR_Handler, NULL);
    XTmrCtr_SetOptions(&timer_controller, 0, XTC_INT_MODE_OPTION | X
```