Tutorial

Contents

General

SOEM is a library that lets your application send and receive EtherCAT frames. It does not contain any logic for maintaining the EtherCAT network in an operational state; however, it has all the building blocks for doing so. In SOEM, it is the user application's responsibility to handle these tasks:

- Reading and writing process data.
- Keeping local I/O data synchronized with the global I/O map.
- Detecting errors reported by SOEM.
- Managing errors reported by SOEM.

The following sections provide basic examples. They show how to start SOEM and how to use process data and check for errors. It is expected that the user modifies these examples to suit their own application.

Creating the context

SOEM requires a context that holds the protocol state. This context must be supplied to all SOEM functions. The context is defined by the application:

```
static ecx_contextt ctx;
```

Configuration

After your application starts, you must set up the NIC. This NIC will be the EtherCAT Ethernet interface. For a simple setup, call `ecx_init()`. If your application intends to support cable redundancy, call `ecx_init_redundant()` instead. This opens a second port as a backup. The function returns a value greater than 0 on success:

```
if (ecx_init(&ctx, iface) <= 0)
{
   /* ERROR */
}
```

Next, you should configure the network. SOEM is a lightweight EtherCAT master library intended for embedded systems. As such, it only supports runtime configuration. Configuration is triggered by calling `ecx_config_init()`:

```
if (ecx_config_init(&ctx) <= 0)
{
   /* ERROR */
}
```

SOEM will enumerate and initialize all slaves found on the network. When initialization finishes, SOEM requests all slaves to enter the PreOP state. The return value is the number of slaves found and should be at least one if enumeration was successful.

SOEM has now discovered and configured the network. You can verify that all expected slaves are present by verifying that the contents of the `ecx_context` member `slavelist` are as expected. This array contains discovered information about each slave:

```
#define EK1100_1           1
#define EL4001_1           2
[...]
#define EL2622_3           8
#define EL2622_4           9
#define NUMBER_OF_SLAVES   9

uint32 verify_network_configuration(ecx_contextt * ctx)
{
   /* Do we have the expected number of slaves from config? */
   if (ctx->slavecount < NUMBER_OF_SLAVES)
      return 0;

   /* Verify each slave */
   if (strcmp(ctx->slavelist[EK1100_1].name, "EK1100"))
      return 0;
   else if (strcmp(ctx->slavelist[EL4001_1].name, "EL4001"))
      return 0;
   [...]
   else if (strcmp(ctx->slavelist[EL2622_4].name, "EL2622"))
      return 0;

  return 1;
}
```

The next step is to map the slaves into the IOmap by calling `ecx_config_map()`. The function returns the size of the map, and you should check that it does not overflow the supplied buffer:

```
uint8_t IOmap[IOMAP_SIZE];
size = ecx_config_map(&ctx, IOmap);
if (size > IOMAP_SIZE)
{
   /* ERROR */
}
```

During this call, SOEM also calculates the working counter for this network. The working counter indicates how many slaves responded to an EtherCAT frame and is the primary way to detect errors on the network. The application should maintain the expected working counter internally so that communication failures can be detected. It is calculated according to this formula:

```
expectedWKC = ctx.group[0].outputsWKC * 2 + ctx.group[0].inputsWKC;
```

At this point the application should also call `ecx_configdc()` to configure the EtherCAT clock and measure propagation delays of all slaves:

```
ecx_configdc(&ctx);
```

After mapping a slave, SOEM requests it to enter the SafeOP state. The application should wait until all slaves on the network enter the requested state by calling `ecx_statecheck()`:

```
ecx_statecheck(&ctx, 0, EC_STATE_SAFE_OP, EC_TIMEOUTSTATE * 4);
```

Before entering operational state, the application should first ensure slaves have valid outputs by sending process data. Process data is sent and received using the functions `ecx_send_processdata()` and `ecx_receive_processdata()`:

```
ecx_send_processdata(&ctx);
ecx_receive_processdata(&ctx, EC_TIMEOUTRET);
```

Finally, the operational state can be requested through `ecx_writestate()` using slave 0, which will broadcast the request to all slaves on the network:

```
ctx.slavelist[0].state = EC_STATE_OPERATIONAL;
ecx_writestate(&ctx, 0);
```

The application should then check that all slaves could enter operational state:

```
chk = 200;
/* wait for all slaves to reach OP state */
do
{
   ecx_send_processdata(&ctx);
   ecx_receive_processdata(&ctx, EC_TIMEOUTRET);
   ecx_statecheck(&ctx, 0, EC_STATE_OPERATIONAL, 50000);
} while (chk-- && (ctx.slavelist[0].state != EC_STATE_OPERATIONAL));
if (ctx.slavelist[0].state != EC_STATE_OPERATIONAL)
{
   /* ERROR */
}
```

Once operational state has been reached, the application must continue to send process data. Failure to do so in a timely manner is likely to trigger slave watchdogs, which will cause the slaves to go back to the SafeOP state. Sending and receiving process data can be done in a simple loop:

```
for (;;)
{
   os_usleep(5000);

   ecx_send_processdata(&ctx);
   wkc = ecx_receive_processdata(&ctx, EC_TIMEOUTRET);
   if (wkc != expectedWKC)
   {
      /* ERROR */
   }
}
```

> **Tip**
>
> See this how-to if you plan to perform XoE (CoE, FoE, EoE, etc) commands from other threads.

Some applications may need to control the jitter of the cyclic loop. One method could be to synchronize an internal clock with the EtherCAT distributed clock.

> **Tip**
>
> See this how-to for guidance on how to use distributed clocks effectively.

Processdata access

The application accesses the process data through the IOmap. In a C program, a simple method for working with the IOmap is to overlay a struct on the buffer. The members of the struct correspond to the inputs and outputs and can be easily accessed by dereferencing the struct.

The definition of the struct will depend on the mapping. You can use the **slaveinfo** sample program with the `map` parameter to dump the mapping SOEM has produced for a given network.

```
slaveinfo <ifname> -map
```

```
Slave:1
[...]
PDO mapping according to CoE :
  SM2 outputs
     addr b   index: sub bitl data_type    name
  [0x0000.0] 0x7000:0x01 0x08 UNSIGNED8    LED0
  [0x0001.0] 0x7001:0x01 0x08 UNSIGNED8    LED1
  SM3 inputs
     addr b   index: sub bitl data_type    name
  [0x0004.0] 0x6000:0x01 0x08 UNSIGNED8    Button1

Slave:2
[...]
PDO mapping according to CoE :
  SM2 outputs
     addr b   index: sub bitl data_type    name
  [0x0002.0] 0x7100:0x00 0x08 UNSIGNED8    Output 8 bits
  [0x0003.0] 0x7200:0x00 0x08 UNSIGNED8    Output 8 bits
  SM3 inputs
     addr b   index: sub bitl data_type    name
  [0x0005.0] 0x6000:0x00 0x08 UNSIGNED8    Input 8 bits
  [0x0006.0] 0x6200:0x00 0x08 UNSIGNED8    Input 8 bits
```

The **slaveinfo** output shows that the first four bytes are the outputs of slave 1 and slave 2, and that the inputs of those slaves then follow. This corresponds to the following C struct:

```
OSAL_PACKED_BEGIN
typedef struct OSAL_PACKED
{
   struct
   {
      uint8_t LED0;
      uint8_t LED1;
      uint8_t O1;
      uint8_t O2;
   } outputs;
   struct
   {
      uint8_t Button1;
      uint8_t I1;
      uint8_t I2;
   } inputs;
} processdata_t;
OSAL_PACKED_END

processdata_t *pd = (processdata_t *)IOmap;
```

to be used as:

```
pd->outputs.LED0 = 1;
pd->outputs.LED1 = 0;
buttonState = pd->inputs.Button1;
```

For bit-oriented slaves, you may need to use bit-fields to correctly represent all process data members.

Packed mode

In non-packed mode, process data for each slave begins at a byte boundary. This may simplify access of process data for some slaves.

In packed mode, process data for a slave may not start at a byte boundary. This may result in smaller process data frame sizes.

Non-packed mode is the default. Enable packed mode by setting `packedMode` to `TRUE` in the `ecx_context` struct:

```
static ecx_contextt ctx = {
   .packedMode = TRUE,
};
```

Overlapped mode

In non-overlapped mode, outgoing process data frames on the bus are composed of outputs followed by space reserved for inputs. Slaves on the network will fill in inputs in the returning frame.

In overlapped mode, outgoing process data frames contain only outputs. Slaves on the network will replace outputs with inputs in the return frame. This mode must be used with some EtherCAT Slave Controllers, notably those from Texas Instruments.

Non-overlapped mode is the default. Enable overlapped mode by setting `overlappedMode` to `TRUE` in the `ecx_context` struct:

```
static ecx_contextt ctx = {
   .overlappedMode = TRUE,
};
```

Custom configuration

You can set a configuration hook to be run for a particular slave on the network. For instance, some slaves may need parameterization before being configured. The hook must be configured before calling `ecx_config_map()`. It will run at the state transition between PreOP and SafeOP:

```
int EL7031setup(ecx_contextt * ctx, uint16 slave)
{
   int retval;
   uint16 u16val;

   retval = 0;

   /* Map velocity PDO assignment via Complete Access */
   uint16 map_1c12[4] = {0x0003, 0x1601, 0x1602, 0x1604};
   uint16 map_1c13[3] = {0x0002, 0x1a01, 0x1a03};

   retval += ecx_SDOwrite(ctx, slave, 0x1c12, 0x00, TRUE, sizeof(map_1c12), &map_1c
   retval += ecx_SDOwrite(ctx, slave, 0x1c13, 0x00, TRUE, sizeof(map_1c13), &map_1c

   /* Set some motor parameters, as an example */
   u16val = 1200; // max motor current in mA
   retval += ecx_SDOwrite(ctx, slave, 0x8010, 0x01, FALSE, sizeof(u16val), &u16val,
   u16val = 150; // motor coil resistance in 0.01ohm
   retval += ecx_SDOwrite(ctx, slave, 0x8010, 0x04, FALSE, sizeof(u16val), &u16val,

   /* Set other necessary parameters as needed */

   return 1;
}
```

Install the hook by identifying the slave and setting its PO2SOconfig parameter:

```
for (i = 1; i <= ctx.slavecount; i++)
{
   ec_slavet * slave = &ctx.slavelist[i];

   if (slave->eep_man == 0x00000002 && slave->eep_id == 0x1b773052)
   {
      slave->PO2SOconfig = EL7031setup;
   }
}

ecx_config_map_group(&ctx, IOmap, 0);
```

EtherCAT slave groups

EtherCAT slaves can be assigned to separate logical groups. Each group has its own logical address space. This space maps to an IOmap address. It allows sending and receiving process data at different update rates.

A slave is assigned to a group by setting its group number.

> **Note**
>
> A slave can only belong to one group.

In this example, slaves are assigned to two different groups:

```
for (i = 1; i <= ctx.slavecount; i++)
{
   ec_slavet * slave = &ctx.slavelist[i];

   if ( <some condition> )
   {
      slave->group = 1;
   }
   else
   {
      slave->group = 2;
   }
}
```

You must then configure each group by calling `ecx_config_map_group()` and supplying the IOmap for that group:

```
ecx_config_map_group(&ctx, IOmap1, 1);
ecx_config_map_group(&ctx, IOmap2, 2);
```

To exchange process data, call `ecx_send_processdata_group()` and `ecx_receive_processdata_group()` for each group:

```
ecx_send_processdata_group(&ctx, 1);
ecx_receive_processdata_group(&ctx, 1, TIMEOUT_GROUP1);

ecx_send_processdata_group(&ctx, 2);
ecx_receive_processdata_group(&ctx, 2, TIMEOUT_GROUP2);
```

Error handling

The working counter is the main mechanism for detecting slave errors. Each slave that receives an EtherCAT frame acknowledges that it has handled it successfully by updating the working counter. If the working counter of a returning frame is not equal to the expected working counter, this indicates that one or more slaves did not process the outgoing frame successfully.

An error handler should check the state of all slaves on the network and attempt to recover any slave that is not in the expected state:

```
if (inOP && ((dowkccheck > 2) || ctx.grouplist[currentgroup].docheckstate))
{
   /* one or more slaves are not responding */
   ctx.grouplist[currentgroup].docheckstate = FALSE;
   ecx_readstate(&ctx);
   for (slaveix = 1; slaveix <= ctx.slavecount; slaveix++)
   {
      ec_slavet *slave = &ctx.slavelist[slaveix];

      if ((slave->group == currentgroup) && (slave->state != EC_STATE_OPERATIONAL))
      {
         ctx.grouplist[currentgroup].docheckstate = TRUE;
         if (slave->state == (EC_STATE_SAFE_OP + EC_STATE_ERROR))
         {
            printf("ERROR : slave %d is in SAFE_OP + ERROR, attempting ack.\n", sla
            slave->state = (EC_STATE_SAFE_OP + EC_STATE_ACK);
            ecx_writestate(&ctx, slaveix);
         }
         else if (slave->state == EC_STATE_SAFE_OP)
         {
            printf("WARNING : slave %d is in SAFE_OP, change to OPERATIONAL.\n", sl
            slave->state = EC_STATE_OPERATIONAL;
            if (slave->mbxhandlerstate == ECT_MBXH_LOST) slave->mbxhandlerstate = E
            ecx_writestate(&ctx, slaveix);
         }
         else if (slave->state > EC_STATE_NONE)
         {
            if (ecx_reconfig_slave(&ctx, slaveix, EC_TIMEOUTMON) >= EC_STATE_PRE_OP
            {
               slave->islost = FALSE;
               printf("MESSAGE : slave %d reconfigured\n", slaveix);
            }
         }
         else if (!slave->islost)
         {
            /* re-check state */
            ecx_statecheck(&ctx, slaveix, EC_STATE_OPERATIONAL, EC_TIMEOUTRET);
            if (slave->state == EC_STATE_NONE)
            {
               slave->islost = TRUE;
               slave->mbxhandlerstate = ECT_MBXH_LOST;
               /* zero input data for this slave */
               if (slave->Ibytes)
               {
                  memset(slave->inputs, 0x00, slave->Ibytes);
               }
               printf("ERROR : slave %d lost\n", slaveix);
            }
         }
      }
      if (slave->islost)
      {
         if (slave->state <= EC_STATE_INIT)
         {
            if (ecx_recover_slave(&ctx, slaveix, EC_TIMEOUTMON))
            {
               slave->islost = FALSE;
               printf("MESSAGE : slave %d recovered\n", slaveix);
            }
         }
         else
         {
            slave->islost = FALSE;
            printf("MESSAGE : slave %d found\n", slaveix);
         }
      }
   }
   if (!ctx.grouplist[currentgroup].docheckstate)
      printf("OK : all slaves resumed OPERATIONAL.\n");
   dowkccheck = 0;
}
```

Accessing CoE SDOs and PDOs

Slaves communicate with the master using several methods. CANopen over EtherCAT (CoE) is a slow but flexible mechanism. It transfers data using a client/server approach via mailboxes. Each transaction typically takes several EtherCAT frames.

SOEM provides `ecx_SDOread()` and `ecx_SDOwrite()` functions for accessing CoE SDOs.

See this how-to if you plan to perform CoE or any other XoE protocol command from multiple threads.

SOEM does not have specific functions for CoE PDOs (Process Data Objects). However, you can often use the same SDO functions. This works on most slaves. In rare cases, the PDO object is marked as “PDO only” in the CoE dictionary. Then, only IOmap access is allowed.