Sistemas Embebidos II

Semestre de Verão de 2010/2011

Quarta atividade prática

eCos - Ethernet Device Driver

Na terceira parte da segunda atividade prática, foi realizado um gestor de periférico, para rede Ethernet, sobre controlador ENC28J60, com capacidade para armazenamento dos pacotes a enviar e dos pacotes recebidos.

O objetivo desta atividade prática é adaptar esse gestor de periférico ao modelo de programação eCos.

O gestor deve exportar a interface seguinte:

```
void ethernet_init(cyg_uint8 * mac);

int ethernet_send(cyg_uint8 * packet, size_t packet_size,
      cyg_tick_count_t timeout);

size_t ethernet_recv(cyg_uint8 * buffer, size_t buffer_size,
      cyg_tick_count_t timeout);
```

A *thread* utilizadora deve ficar em espera passiva se, ao invocar **`ethernet_recv`**, não hajam pacotes recebidos ou, ao invocar **`ethernet_send`**, não haja espaço no *buffer* de saída.

O parâmetro **`timeout`** serve para especificar, em tiques do relógio do eCos, o máximo tempo de espera pela realização da operação requerida.

A adaptação vai incidir no processamento das interrupções e na comunicação SPI. Quanto ao processamento de interrupções, deve aplicar os conhecimentos já adquiridos na terceira atividade prática. Quanto à comunicação SPI, deve passar a utilizar o suporte existente no eCos, documentado aqui: http://ecos.sourceware.org/docs-latest/ref/io-spi.html.

Para testar a comunicação SP, aconselha-se a utilização da primeira versão do gestor de comunicação Ethernet, sem atendimento por interrupção, substituindo o suporte SPI desenvolvido na primeira atividade prática, pelo suporte eCos. As filosofias das interfaces de programação são idênticas.

O dispositivo SPI é representado por uma variável que contém os parâmetros do protocolo e a função de manipulação do *chip select*. No exemplo que se segue a variável **`spi_enc28j60_dev`** tem esse propósito.

```
static void spi_enc28j60_cs(int select) {
      if (select)
            HAL_WRITE_UINT32(CYGARC_HAL_LPC2XXX_REG_IO_BASE +
                  CYGARC_HAL_LPC2XXX_REG_IOCLR, 1 << CS_PIN);
      else
            HAL_WRITE_UINT32(CYGARC_HAL_LPC2XXX_REG_IO_BASE +
                  CYGARC_HAL_LPC2XXX_REG_IOSET, 1 << CS_PIN);
}
```

```
cyg_spi_lpc2xxx_dev_t spi_enc28j60_dev CYG_SPI_DEVICE_ON_BUS(0) = {
      .spi_device.spi_bus = &cyg_spi_lpc2xxx_bus0.spi_bus,
      .spi_cpha   = 0,        // Clock phase (0 or 1)
      .spi_cpol   = 0,        // Clock polarity (0 or 1)
      .spi_lsbf   = 0,        // MSBF
      .spi_baud   = 1000000,  // Clock baud rate
      .spi_cs     = spi_enc28j60_cs
};
```

Note as semelhanças entre a função seguinte e a usada na primeira atividade prática.

```
#define SPI_DEV   (cyg_spi_device *)&spi_enc28j60_dev

static void read_buffer_memory(cyg_uint8 * buffer, size_t size) {
      cyg_uint8 tx_buf[] = { 0x3a };
      cyg_spi_transaction_begin(SPI_DEV);
      cyg_spi_transaction_transfer(SPI_DEV, true, 1, tx_buf, NULL, false);
      cyg_spi_transaction_transfer(SPI_DEV, true, size, buffer, buffer, true);
      cyg_spi_transaction_end(SPI_DEV);
}
```