# eCos

Kernel

HAL

Debug

Bibliotecas normalizadas POSIX.

Bibliotecas diversas: SPI, RTC, CAN

TCP/IP

Sistemas de ficheiros (em FLASH)

Graficos

USB

# Geração da aplicação

A integração da aplicação com o sistema operativo eCos processa-se por ligação estática. O eCos apresenta-se como um conjunto de módulos compilados e agrupados em biblioteca – **libtarget.a**.

Os programas de aplicação organizam-se também em módulos compilados separadamente com base nas declarações das interfaces exportadas pelo eCos.

A biblioteca eCos **libtarget.a**, assim como os ficheiros de inclusão com as respectivas declarações, são depositados numa sub-árvore como a que a seguir se apresenta. Os ficheiros depositados nesta sub-árvore são dependentes da arquitectura do processador, do *hardware* envolvente e de escolhas de configuração.

```
install
    ecos_build
    ecos_install
        include
            cyg
                kernel
                    kapi.h
                    ...
            pkconf
                ...
            sys
        lib
            libtarget.a
            target.ld
    ecos.ecc
```

Esta sub-árvore é gerada pela ferramenta de configuração **configtool**. A directoria **`ecos_build`** é usada para depositar temporariamente os ficheiros objecto resultantes da compilação separada. O ficheiro **`ecos.ecc`** contém o registo das opções escolhidas em **configtool**.

O pacote ecos, na directoria **`examples`**, contém um *script* para gerar o ficheiro **`Makefile`** a usar na geração de aplicações. Para o utilizar deve posicionar-se na directoria onde tem os ficheiros da aplicação, definir as variáveis de ambiente **`SRCS`** e **`DST`** e executá-lo.

Exemplo:

```
se2
    ecos
        app
            hello
        mylib
        ecos-3.0
        install
        ...

$ cd ${SE2}/ecos/app/hello
$ export SRCS=main.c
$ export DST=main
$ ${SE2}/ecos/ecos-3.0/examples/build_Makefile \
    ${SE2}/ecos/ecos_install_dir
```

Foi criado o seguinte ficheiro **`Makefile`** com os caminhos e as opções de geração adequadas.

**`Makefile`**

```
#
# Makefile for eCos tests
#

# Platform specific setups
include Make.params

# Simple build rules

.c.o:
    $(CC) -c $(ACTUAL_CFLAGS) -I$(PREFIX)/include $*.c

.o:
    $(CC) $(ACTUAL_LDFLAGS) -L$(PREFIX)/lib -Ttarget.ld $*.o -o $@

SRCS=main.c
OBJS=${SRCS:%.c=%.o}
DST=main

${DST}: ${OBJS}
```

**`main.c`**

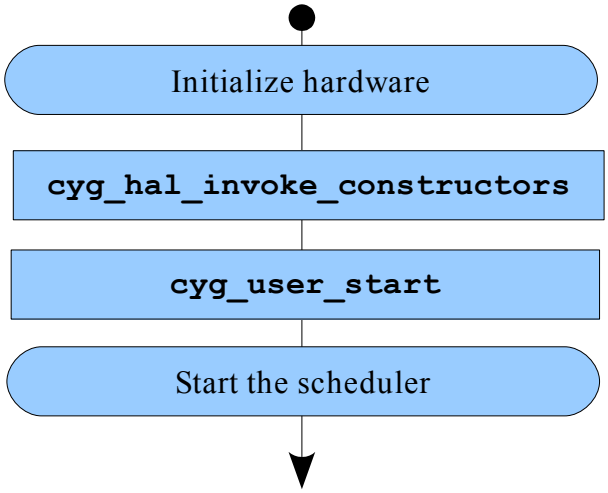
```
#include <stdio.h>

int main(void)
{
  printf("Hello, eCos world!\n");
  return 0;
}
```

# Sequência de iniciação do eCos

O eCos é escrito em **C++**, no entanto a sua interface externa é **C**. A sua iniciação é feita em construtores de objectos estáticos. Estes construtores são executados a partir da função **`cyg_hal_invoke_constructors`**. A ordem de execução é garantida através de um esquema de prioridades.

O programa do utilizador pode ter início na função **`main`** ou na função **`cyg_user_start`.** O método preferível para iniciar a aplicação (criação de *threads* e outros objectos do núcleo) é a partir de **`cyg_user_start`**. Nesta altura a comutação de

tarefas ainda está desactivada.




A função **`main`** é executada como uma *thread.* Esta *thread* é criada no construtor do objecto estático **`cyg_libc_main_thread.`**

Os construtores dos objectos estáticos só podem usar recursos já iniciados em estágios de maior prioridade, já executados.

O eCos fornece as funções **`main`** ou **`cyg_user_start`** vazias para o caso de o utilizador não as definir.

# Threads

Os dados específicos de uma *thread* são mantidos pelo núcleo numa estrutura de dados designada por "descritor da *thread"*. Fazem parte dos dados específicos duma *thread* o estado, a prioridade e o contexto de execução.

Os estados fundamentais de uma *thread* são: a executar, pronta a executar ou bloqueada. O bloqueio pode ser devido a uma temporização, espera que um recurso esteja disponível, espera que ocorra determinado evento.

O identificador de uma *thread* refere-se, normalmente de forma indirecta, ao seu descritor.

## *Criação*

```
void cyg_thread_create(
    cyg_addrword_t sched_info,
    cyg_thread_entry_t * function_entry,
    cyg_addrword_t entry_data,
    char *name,
    void *stack_base,
    cyg_ucount32 stack_size,
    cyg_handle_t *handle,
    cyg_thread *thread
);
```

Prioridade
Contexto da thread
Nome legível
Para identificação da thread
Descritor da thread

As *threads* têm prioridade fixa. A um valor menor corresponde uma prioridade maior.

O código de uma *thread* deve ser especificado numa função com a seguinte forma:

```
void thread_function_entry(cyg_addrword_t data) { ... }
```

Na criação de uma *thread*, além do código, é também especificado um elemento de informação – **data**, que será passado como argumento à *thread*. Isto permite usar o mesmo código em várias *threads* a processar sobre contextos diferentes.
A operação de criação de uma *thread* sucede sempre bem, porque o programa criador fornece os recursos necessários para a sua criação: **thread_stack** – memória para *stack* e **thread_obj** - memória para o descritor.
O identificador da *thread* usado nas restantes operações é depositado na variável de saída **handle**.
As *threads* nascem no estado suspenso.

**Exemplo**

```
#include <cyg/hal/hal_arch.h>       /* For stack size */
#include <cyg/kernel/kapi.h>        /* Kernel API */
#include <cyg/infra/diag.h>         /* For diagnostic printing */

#define THREAD_STACK_SIZE           CYGNUM_HAL_STACK_SIZE_TYPICAL
#define THREAD_PRIORITY             12

int thread_stack[THREAD_STACK_SIZE];

cyg_thread thread_obj;

cyg_handle_t thread_hdl;

void thread(cyg_addrword_t data) {
    diag_printf("thread starting=%d\n", data);
    while ( 1 ) {
        diag_printf("thread running\n");
    }
}

void cyg_user_start( void ) {

    diag_printf("Hello eCos World!!!\n\n");

    cyg_thread_create(THREAD_PRIORITY, thread, (cyg_addrword_t) 75,
        "Thread name", (void *)thread_stack, THREAD_STACK_SIZE,
        &thread_hdl, &thread_obj);

    cyg_thread_resume (thread_hdl);
}
```

## Terminação

Terminar a execução da *thread* evocante removendo-a do *scheduler*. Não liberta recursos, a *thread* continua a existir apenas não executa. Permite continuar a executar algumas operações sobre a thread.

```
void cyg_thread_exit();
```

Produz na thread referida um efeito semelhante a **cyg_thread_exit**. É perigoso porque a thread alvo pode deter recuros que não serão libertados.

```
void cyg_thread_kill(cyg_handle_t thread);
```

Elimina completamente a *thread* visada. A partir daqui a memória de *stack* pode ser reutilizada e o *handle* já não é válido.

```
cyg_bool_t cyg_thread_delete(cyg_handle_ thread);
```

## *Stack* da *thread*

O *stack* serve para alojar dados locais e para manter o registo da sequência de chamadas.

O espaço para *stack* é fornecido pelo criador da nova *thread.*

A dimensão de *stack* necessária depende do número de chamadas encadeadas, da dimensão dos objectos locais, do critério de utilização quanto ao protocolo de chamadas a funções, da possibilidade de execução de rotinas de interrupção, etc.

Dimensão mínima para *stack* de uma *thread*.

**`CYGNUM_HAL_STACK_SIZE_MINIMUM`** (não configurável em configtool)

Dimensão normal para o *stack* de uma *thread.*

**`CYGNUM_HAL_STACK_SIZE_TYPICAL`** (não configurável em configtool)

Utilização de *stack* separado para a execução das rotinas de interrupção.

**`CYGIMP_HAL_COMMON_INTERRUPTS_USE_INTERRUPT_STACK`**

Permitir o processamento aninhado de interrupções, isto é, permitir que uma rotina de interrupção seja interrompida para dar lugar à execução de outra rotina de maior prioridade.

**`CYGSEM_HAL_COMMON_INTERRUPTS_ALLOW_NESTING`**

O *stack* é preenchido com um padrão e em cada comutação de *thread* é medida a dimensão de *stack* usada até à altura.

**`CYGFUN_KERNEL_THREADS_STACK_MEASUREMENT`**

## Informação da thread

Obter o *handle* da *thread* corrente.

```
cyg_handle_t cyg_thread_self();
```

Obter o *handle* da *thread idle.*

```
cyg_handle_t cyg_thread_idle_thread();
```

Obter o endereço base do *stack.*

```
cyg_addrword_t cyg_thread_get_stack_base(cyg_handle_t thread);
```

Obter a dimensão do *stack*

```
cyg_uint32 cyg_thread_get_stack_size(cyg_handle_t thread);
```

Obter a dimensão de *stack* usado

```
cyg_uint32 cyg_thread_measure_stack_usage(cyg_handle_t thread);
```

Para enumerar as *threads* existentes.

```
cyg_bool cyg_thread_get_next(cyg_handle_t *thread, cyg_uint16 *id);
```

Obter informação de uma *thread.*

```
cyg_bool cyg_thread_get_info(
    cyg_handle_t thread, cyg_uint16 id, cyg_thread_info * info);
```

Obter o identificador dado o *handle* da *thread.*

```
cyg_handle_t cyg_thread_get_id(cyg_handle_t thread);
```

Obter o *handle* da *thread* identificada por id.

```
cyg_handle_t cyg_thread_find(cyg_uint16 id);
```

```
typedef struct {
    cyg_handle_t handle;
    cyg_uint16 id;
    cyg_uint32 state;
```

```
        char *name;
        cyg_priority_t set_pri;
        cyg_priority_t cur_pri;
        cyg_addrword_t stack_base;
        cyg_uint32 stack_size;
        cyg_uint32 stack_used;
    } cyg_thread_info;
```

**Exemplo**

```
#include <cyg/kernel/kapi.h>
#include <stdio.h>

void show_threads(void) {
    cyg_handle_t thread = 0;
    cyg_uint16 id = 0;
    while (cyg_thread_get_next(&thread, &id)) {
        cyg_thread_info info;
        if ( ! cyg_thread_get_info( thread, id, &info))
            break;
        diag_printf("ID: %04x name: %10s pri: %d\n",
            info.id, info.name? info.name: "----", info.set_pri );
    }
}
```

## Controlo da thread

Pausa temporizada.

```
void cyg_thread_delay(cyg_tick_count_t delay);
```

Suspender a execução de uma tarefa.

```
void cyg_thread_suspend(cyg_handle_t thread);
```

Retomar a execução de uma tarefa suspensa.

```
void cyg_thread_resume(cyg_handle_t thread);
```

Ceder o processador

```
void cyg_thread_yield(void);
```

Cessar uma espera.

```
void cyg_thread_release(cyg_handle_t thread);
```

## Prioridades das threads

O valor da prioridade de uma tarefa vai de 0 ao número a que corresponde o símbolo seguinte, normalmente 31, **`CYGNUM_KERNEL_SCHED_PRIORITIES.`**
Ao valor 0 corresponde a prioridade mais elevada. Á *thread idle* é atribuído o valor mais elevado que corresponde à menor prioridade.

Obter a prioridade base.

```
cyg_priority_t cyg_thread_get_priority(cyg_handle_t thread);
```

Obter a prioridade actual. Pode ser alterada para evitar inversão de prioridade.

```
cyg_priority_t cyg_thread_get_current_priority(cyg_handle_t thread);
```

Definir uma nova prioridade base.

```
void cyg_thread_set_priority(cyg_handle_t thread, cyg_priority_t priority);
```

## Dados da thread

Uma *thread* pode ter dados de utilizador associados. Este mecanismo permite programar bibliotecas *thread safe.* Por exemplo a função **`strtok`**.

Alocar uma nova posição.

```
cyg_ucount32 cyg_thread_new_data_index();
```

Libertar uma posição.

```
void cyg_thread_free_data_index( cyg_ucount32 index);
```

Obter os dados de uma posição.

```
CYG_ADDRWORD cyg_thread_get_data( cyg_ucount32 index);
```

Obter o apontador para uma posição.

```
CYG_ADDRWORD cyg_thread_get_data_ptr(cyg_ucount32 index);
```

Definir os dados de uma posição.

```
void cyg_thread_set_data(cyg_ucount32 index, CYG_ADDRWORD data);
```

## Terminadores da thread

As funções destrutoras são invocadas na terminação de uma *thread* ao retornar da função da thread na chamada a **`cyg_thread_exit`** . São usadas para libertar recursos.

Adicionar um destrutor.

```
cyg_bool_t cyg_thread_add_destructor(
    cyg_thread_destructor_fn, cyg_addrword_t data);
```

Remover um destrutor.

```
cyg_bool_t cyg_thread_rem_destructor(
    cyg_thread_destructor_fn, cyg_addrword_t data);
```

## Escalonador de *threads* (scheduler)

O escalonamento é o processo de escolha da próxima *thread* a executar de entre as *threads* prontas a executar. A atribuição de prioridades ás tarefas permite, numa situação de várias tarefas prontas a executar, escolher a mais importante. Ficando ao critério do programador a sua classificação.

### Preensão

Um núcleo de multiprogramação preensivo caracteriza-se pela capacidade de executar a comutação de tarefas assincronamente, isto é, uma tarefa pode ser preterida em qualquer ponto, mesmo sem ter chamado uma função do sistema.
A interrupção é o mecanismo básico de suporte à preensão. Na sequência de uma interrupção pode ficar pronta a executar uma nova tarefa e ser realizada a comutação.
Num núcleo não preensivo uma tarefa de maior prioridade pronta a executar só será executada quando a tarefa corrente libertar o processador.

### Tempo repartido (round-robin ou time slice)

Um núcleo de tempo repartido reparte o processador pelas tarefas prontas a executar, segundo um critério de repartição do tempo. O critério mais comum consiste em atribuir uma parcela de tempo igual a cada tarefa. Num núcleo com prioridades só faz sentido aplicar este critério a tarefas com a mesma prioridade.

### bitmap scheduler

- só pode haver uma *thread* de cada prioridade. O número de threads no sistema é limitado pelo número de prioridades.
- proporciona algoritmos mais eficientes.

### multi-level queue scheduler

- permite que várias threads executem à mesma prioridade.

- suporta tempo repartido (timeslicing).
- o algoritmo de tempo repartido só é aplicado quando existe mais do que uma thread da mesma prioridade prontas a executar e todas as thread de maior prioridade não estão prontas a executar.
- se existir só uma thread pronta a executar, ou o critério de tempo partilhado esteja desligado, a thread em execução não será preendida até se bloquear ou explicitamente ceder o processador – CYGSEM_KERNEL_SCHED_TIMESLICE; CYGNUM_KERNEL_SCHED_TIMESLICE_TICKS
- A inserção em filas de espera nos objectos pode ser por ordem de chegada ou com base na prioridade – CYGIMP_KERNEL_SCHED_SORTED_QUEUES.

A função **`cyg_scheduler_start`** é chamada automaticamente e marca o fim da iniciação. A partir daqui o núcleo passa a comutar as tarefas de acordo com a política definida.

```
void cyg_scheduler_start(void);
```

A natureza preensiva do núcleo do eCos pode ser suspensa e retomada por estas funções.

```
void cyg_scheduler_lock(void);
void cyg_scheduler_unlock(void);
```

Uma aplicação comum não deve usar este mecanismo para sincronização.

### *Reentrância*

Num ambiente preensivo as funções passíveis de serem executadas simultaneamente em diversas *threads* devem ser reentrantes.

Uma função reentrante pode ser interrompida em qualquer altura, executada completamente noutro contexto e retomada no ponto onde tinha sido interrompida sem haver corrupção dos dados.

Uma função reentrante aloja os seus dados em registos ou em *stack* e protege os recursos partilhados (dados alojados em posições estáticas ou periféricos).
A protecção consiste em utilizar meios de sincronização com as outras *threads* de modo a garantir o acesso sequencial aos recurso partilhados.

## Threads e C++

A utilização de *rappers* C++ pode facilitar a programação reduzindo o código escrito.

```
class Thread {
    enum { THREAD_STACK_SIZE = CYGNUM_HAL_STACK_SIZE_TYPICAL };
    enum { THREAD_PRIORITY = 12 };

    int thread_stack[ THREAD_STACK_SIZE ];

    cyg_thread thread_obj;
    cyg_handle_t thread_hdl;

public:
    Thread() {
        cyg_thread_create(
                THREAD_PRIORITY,
                &call_run, (cyg_addrword_t) this,
                "Thread name",
                (void *)thread_stack, THREAD_STACK_SIZE,
               &thread_hdl, &thread_obj );
    }

    virtual void run() {}
```

```
    static void call_run(cyg_addrword_t objptr) {
        Thread * obj = (Thread *) objptr;
        obj->run();
    }
    void delay(int x) { cyg_thread_delay(x); }
    void resume()     { cyg_thread_resume( thread_hdl ); }
};
```

```
#include <cyg/infra/diag.h>
#include <cyg/hal/hal_io.h>
#include <mylib/thread.h>

class My_thread : public Thread {
    int led;
    int time;
public:
    My_thread(int l, int t) : led(l), time(t) { resume(); }
    void run() {
        while( true ) {
            int aux;
            delay(CYGNUM_HAL_RTC_DENOMINATOR * time);
            HAL_READ_UINT32(CYGARC_HAL_LPC2XXX_REG_IO_BASE +
                CYGARC_HAL_LPC2XXX_REG_IOPIN, aux);
            HAL_WRITE_UINT32(CYGARC_HAL_LPC2XXX_REG_IO_BASE +
                CYGARC_HAL_LPC2XXX_REG_IOSET, aux | (1 << led) );

            delay(CYGNUM_HAL_RTC_DENOMINATOR * time);
            HAL_READ_UINT32(CYGARC_HAL_LPC2XXX_REG_IO_BASE +
                CYGARC_HAL_LPC2XXX_REG_IOPIN, aux);
            HAL_WRITE_UINT32(CYGARC_HAL_LPC2XXX_REG_IO_BASE +
                CYGARC_HAL_LPC2XXX_REG_IOCLR, ~aux | (1 << led) );
        }
    }
};

static struct Init_GPIO {
    Init_GPIO() {
        int aux;
        HAL_READ_UINT32(CYGARC_HAL_LPC2XXX_REG_PIN_BASE +
            CYGARC_HAL_LPC2XXX_REG_PINSEL0, aux);
        HAL_WRITE_UINT32(CYGARC_HAL_LPC2XXX_REG_PIN_BASE +
            CYGARC_HAL_LPC2XXX_REG_PINSEL0, aux & ~(3 << 14));
        HAL_WRITE_UINT32(CYGARC_HAL_LPC2XXX_REG_IO_BASE +
            CYGARC_HAL_LPC2XXX_REG_IODIR, 1 << 14 );
        HAL_WRITE_UINT32(CYGARC_HAL_LPC2XXX_REG_IO_BASE +
            CYGARC_HAL_LPC2XXX_REG_IOSET, 1 << 14 );
    }
} init_GPIO CYGBLD_ATTRIB_INIT_BEFORE( CYG_INIT_DEFAULT );

static My_thread thread1(14, 1);
```

# Meios de sincronização

Para sincronizar as *threads* entre si existem os seguintes mecanismos **semaphore**, **mutex**, **condition variables** e **event flags.**

Para sincronizar *threads* com acontecimentos externos existe o objecto **interrupt**.

Para troca de  informação existe o mecanismo **message box.** Este mecanismo integra sincronização e manipulação de dados.

## Mutex

Permite serializar o acesso a um recurso (memória ou dispositivo).

Um mutex é semelhante a um semáforo binário.

Só a thread que fechou o mutex o poderá abrir.

Um mutex fechado não pode ser novamente fechado pela mesma thread.

Uma thread que tente fechar um mutex fechado fica bloqueada até este ser novamente aberto pela thraed que o fechou.

O mutex dispõe de duas soluções para resolver o problema de inversão de prioridades

Priority ceiling protocol – é definida previamente a prioridade a que qualquer thread que feche o mutex é executada (muitas desvantagens).

Priority inheritance protocol – a thread que fechou o mutex é executada à maior prioridade das threads que aguardam a abertura do mutex.

A utilização de um protocolo de inversão de prioridade pode ser desactivada

A escolha do protocolo de inversão de prioridade pode ser feita em execução.

```
CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERTION_PROTOCOL_INHERIT
CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERTION_PROTOCOL_CEILING
CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERTION_PROTOCOL_NONE
CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERTION_PROTOCOL_DEFAULT
```

## API

Iniciar um mutex no estado aberto.

```
void cyg_mutex_init(cyg_mutex_t * mutex);
```

Destruir um mutex.

```
void cyg_mutex_destroy(cyg_mutex_t * mutex);
```

Fechar um mutex.

```
cyg_bool_t cyg_mutex_lock(cyg_mutex_t * mutex);
```

Tentar fechar um mutex.

```
cyg_bool_t cyg_mutex_trylock(cyg_mutex_t * mutex);
```

Abrir um mutex.

```
void cyg_mutex_unlock(cyg_mutex_t* mutex);
```

Libertar todas as threads aguardando no mutex.

```
void cyg_mutex_release(cyg_mutex_t* mutex);
```

Definir a prioridade limite

```
void cyg_mutex_set_ceiling(cyg_mutex_t* mutex,cyg_priority_t priority);
```

Definir o protocolo de inversão de prioridade.

```
void cyg_mutex_set_protocol(
     cyg_mutex_t* mutex, enum cyg_mutex_protocol protocol);
```

## Exemplo

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cyg/kernel/kapi.h>

static cyg_thread thread_desc[2];
```

```
static char thread_stack[2][4096];
static cyg_handle_t thread_handle[2];
static cyg_mutex_t mutex;

void simple_thread(cyg_addrword_t data) {
    int message = (int) data;
    int delay;
    printf("Beginning execution; thread data is %d\n", message);
    while (1) {
        delay = 200 + (rand() % 50);
        cyg_mutex_lock(&mutex);
        printf("Thread %d: and now a delay of %d clock ticks\n",
            message, delay);
        cyg_mutex_unlock(&mutex);
        cyg_thread_delay(delay);
    }
}

void cyg_user_start(void) {
    printf("Entering twothreads' cyg_user_start() function\n");
    cyg_mutex_init(&mutex);
    cyg_thread_create(4, simple_thread, (cyg_addrword_t) 0,
        "Thread A", (void *) thread_stack[0], sizeof(thread_stack[0]),
        &thread_handle[0], &thread_desc[0]);
    cyg_thread_create(4, simple_thread, (cyg_addrword_t) 1,
        "Thread B", (void *) thread_stack[1], sizeof(thread_stack[0]),
        &thread_handle[1], &thread_desc[1]);
    cyg_thread_resume(thread_handle[0]);
    cyg_thread_resume(thread_handle[1]);
}
```

## Condition Variables

As condições são usadas em conjunção com os *mutexes* para realizar esperas longas.

Quando uma thread satisfaz a condição acorda uma ou todas as threads que estavam à espera.

A fução cyg_cond_wait executa duas acções atomicamente: liberta o mutex e coloca-se em espera.

O teste da condição e a respectiva espera deve ser realizada com while. Porque a sinalização apenas desbloqueia a thread não adquire o mutex. Este continua na posse da thread sinalizadora.

Antes de poder ser utilizada a condição deve ser associada a um mutex.

Uma thread só pode evocar o mutex depois de ter fechado o mutex.

A chamada a cyg_cond_signal não requer a posse do mutex.

A função cyg_cond_signal acorda a thread à cabeça da fila de espera, enquanto cyg_cond_broadcast acorda todas as thread em espera.

Se não existirem thread à espera a sinalização não tem qualquer efeito nem será memorizada.

cyg_cond_signal deve ser usado quando várias threads testam a mesma condição e apenas uma pode prosseguir.

cyg_cond_broadcast deve ser usado quando uma alteração de estado satisfaz várias condições aguardadas por diferentes threads.

### API

Associar uma condição a um mutex

```
void cyg_cond_init(cyg_cond_t* cond,cyg_mutex_t* mutex);
```

Destruir uma condição

```
void cyg_cond_destroy(cyg_cond_t* cond);
```

Esperar sinalização

```
cyg_bool_t cyg_cond_wait(cyg_cond_t* cond);
```

Esperar sinalização com temporização

```
cyg_bool_t cyg_cond_timed_wait(cyg_cond_t* cond, cyg_tick_count_t abstime);
```

Sinalizar uma condição libertando uma thread

```
void cyg_cond_signal(cyg_cond_t* cond);
```

Sinalizar uma condição libertando todas as threads

```
void cyg_cond_broadcast(cyg_cond_t* cond);
```

## Exemplo

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cyg/kernel/kapi.h>

typedef struct {
    char data[10];
    int put;
    int get;
    int count;
} buffer_t;

static void buffer_write(buffer_t * b, char c) {
    b->data[b->put] = c;
    if (++(b->put) == sizeof(b->data))
        b->put = 0;
    ++(b->count);
}

static char buffer_read(buffer_t * b) {
    char aux = b->data[b->get];
    if(++(b->get) == sizeof(b->data))
        b->get = 0;
    --(b->count);
    return aux;
}

static int buffer_empty(buffer_t * b) {
    return b->count == 0;
}

static int buffer_full(buffer_t * b) {
    return b->count == sizeof(b->data);
}

static cyg_thread thread_desc[2];
static char thread_stack[2][4096];
static cyg_handle_t thread_hdl[2];
static cyg_mutex_t mutex;
static cyg_cond_t full_condition, empty_condition;

static buffer_t buffer;

static void producer(cyg_addrword_t data) {
    char n;
    for (n  = 0; ; ++n) {
```

```
        int delay = 200 + (rand() % 50);
        cyg_mutex_lock(&mutex);
        while (buffer_full(&buffer))
            cyg_cond_wait(&full_condition);
        buffer_write(&buffer, n);
        cyg_cond_signal(&empty_condition);
        cyg_mutex_unlock(&mutex);
        cyg_thread_delay(delay);
    }
}

static void consumer(cyg_addrword_t data) {
    char n;
    while (1) {
        int delay = 200 + (rand() % 50);
        cyg_mutex_lock(&mutex);
        while (buffer_empty(&buffer))
            cyg_cond_wait(&empty_condition);
        n = buffer_read(&buffer);
        cyg_cond_signal(&full_condition);
        cyg_mutex_unlock(&mutex);
        printf("Thread consumer %d\n", n);
        cyg_thread_delay(delay);
    }
}

void cyg_user_start(void) {
    printf( "eCos condition variable example\n\n" );
    cyg_mutex_init(&mutex);
    cyg_cond_init(&empty_condition, &mutex);
    cyg_cond_init(&full_condition, &mutex);
    cyg_thread_create(4, producer, (cyg_addrword_t) 0,
        "Thread consumer", (void *) thread_stack[0], sizeof(thread_stack[0]),
        &thread_hdl[0], &thread_desc[0]);
    cyg_thread_create(4, consumer, (cyg_addrword_t) 1,
        "Thread producer", (void *) thread_stack[1], sizeof(thread_stack[0]),
        &thread_hdl[1], &thread_desc[1]);
    cyg_thread_resume(thread_hdl[0]);
    cyg_thread_resume(thread_hdl[1]);
}
```

## Semaphores

Associado a cada semáforo existe um contador e uma fila de espera.

Cada unidade do semáforo representa uma entidade concreta.

Sobre o semáforo podem fazer-se operações de incremento e de decremento do contador

Se o contador for zero a tentativa de o decrementar leva ao bloqueio da thread.

A operação de incremento cria a condição para desbloquear uma thread.

## API

Iniciar um semaforo com val unidades.

```
void cyg_semaphore_init(cyg_sem_t* sem,cyg_count32 val);
```

Destruir um semaforo.

```
void cyg_semaphore_destroy(cyg_sem_t* sem);
```

Esperar por unidades

```
cyg_bool_t cyg_semaphore_wait(cyg_sem_t* sem);
```

Espera por unidades temporizada

```
cyg_bool_t cyg_semaphore_timed_wait(cyg_sem_t* sem,
    cyg_tick_count_t abstime);
```

Experimentar o semaforo

```
cyg_bool_t cyg_semaphore_trywait(cyg_sem_t* sem);
```

Incrementar as unidades

```
void cyg_semaphore_post(cyg_sem_t* sem);
```

Ler as unidades sem modificar

```
void cyg_semaphore_peek(cyg_sem_t* sem,cyg_count32* val);
```

## Exemplo

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cyg/kernel/kapi.h>

static cyg_thread thread_desc[2];
static char thread_stack[2][4096];
static cyg_handle_t thread_hdl[2];

cyg_sem_t semaphore;

static void producer(cyg_addrword_t data) {
    while (1) {
        int delay = 200 + (rand() % 50);
        cyg_semaphore_post(&semaphore);
        cyg_thread_delay(delay);
    }
}

static void consumer(cyg_addrword_t data) {
    while (1) {
        cyg_semaphore_wait(&semaphore);
        printf("Thread consumer\n");
    }
}

void cyg_user_start(void) {
    printf( "eCos semaphore example\n\n" );
    cyg_semaphore_init(&semaphore, 0);
    cyg_thread_create(4, producer, (cyg_addrword_t) 0,
        "Thread consumer", (void *) thread_stack[0], sizeof(thread_stack[0]),
        &thread_hdl[0], &thread_desc[0]);
    cyg_thread_create(4, consumer, (cyg_addrword_t) 1,
        "Thread producer", (void *) thread_stack[1], sizeof(thread_stack[0]),
        &thread_hdl[1], &thread_desc[1]);
    cyg_thread_resume(thread_hdl[0]);
    cyg_thread_resume(thread_hdl[1]);
}
```

## Event flags

Um objecto Event Flags é baseado numa palavra de 32 bits.

Cada bit é associado a um acontecimento.

Os bits são sinalizados por threads produtoras ou por DSRs.

Os bits podem ser sinalizados individualmente.

As threads consumidoras aguardam a sinalização de acontecimentos, individualmente ou em conjunção.

Serve para uma thread poder receber a sinalização de acontecimentos de origens diversas.

## API

Criar umas novas flags.

```
void cyg_flag_init(cyg_flag_t* flag);
```

Destruir umas flags.

```
void cyg_flag_destroy(cyg_flag_t* flag);
```

Afectar flags com value.

```
void cyg_flag_setbits(cyg_flag_t* flag, cyg_flag_value_t value);
```

Limpar as flags indicadas em value.

```
void cyg_flag_maskbits(cyg_flag_t* flag, cyg_flag_value_t value);
```

Esperar por combinação de flags.

```
cyg_flag_value_t cyg_flag_wait(cyg_flag_t* flag,
    cyg_flag_value_t pattern, cyg_flag_mode_t mode);
```

Espera temporizada.

```
cyg_flag_value_t cyg_flag_timed_wait(cyg_flag_t* flag,
    cyg_flag_value_t pattern, cyg_flag_mode_t mode,
    cyg_tick_count_t abstime);
```

Testar estado das flags.

```
cyg_flag_value_t cyg_flag_poll(cyg_flag_t* flag,
    cyg_flag_value_t pattern, cyg_flag_mode_t mode);
```

Ler estado das flags.

```
cyg_flag_value_t cyg_flag_peek(cyg_flag_t* flag);
```

Verificar se há threads esperando.

```
cyg_bool_t cyg_flag_waiting(cyg_flag_t* flag);
```

## Exemplo

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cyg/kernel/kapi.h>

static cyg_thread thread_desc[3];
static char thread_stack[3][4096];
static cyg_handle_t thread_hdl[3];

#define  EVF0     0
#define  EVF1     1

cyg_flag_t flags;

static void producer(cyg_addrword_t data) {
    cyg_flag_value_t mask = data;
    while (1) {
        int delay = 200 + (rand() % 50);
        cyg_flag_setbits(&flags, mask);
        cyg_thread_delay(delay);
    }
}

static void consumer(cyg_addrword_t data) {
    while (1) {
```

```
        int delay = 200 + (rand() % 55);
        cyg_flag_value_t mask = cyg_flag_wait(&flags,
             (1 << EVF0) | (1 << EVF1),
            CYG_FLAG_WAITMODE_OR | CYG_FLAG_WAITMODE_CLR);
        if (mask & (1 << EVF0))
             printf("EVF0 ");
        if (mask & (1 << EVF1))
             printf("EVF1");
        printf("\n");
        cyg_thread_delay(delay);
    }
}

void cyg_user_start(void) {
    printf( "eCos event flags example\n\n" );
    cyg_flag_init(&flags);
    cyg_thread_create(4, producer, (cyg_addrword_t) 1 << EVF0,
        "Thread producer1", (void *) thread_stack[0], sizeof(thread_stack[0]),
        &thread_hdl[0], &thread_desc[0]);
    cyg_thread_create(4, producer, (cyg_addrword_t) 1 << EVF1,
        "Thread producer2", (void *) thread_stack[1], sizeof(thread_stack[1]),
        &thread_hdl[1], &thread_desc[1]);
    cyg_thread_create(4, consumer, (cyg_addrword_t) 0,
        "Thread consumer", (void *) thread_stack[2], sizeof(thread_stack[2]),
        &thread_hdl[2], &thread_desc[2]);
    cyg_thread_resume(thread_hdl[0]);
    cyg_thread_resume(thread_hdl[1]);
    cyg_thread_resume(thread_hdl[2]);
}
```

## Message Box

Tem uma semântica de sincronização idêntica à do semáforo.

Em vez de um contador possui uma fila de dados.

Uma thread produtora sinaliza enviando um elemento de dados.

A thread consumidora recebe essa sinalização na forma desse elemento de dados.

O elemento de dados é um apontador para uma zona de memória que transporta uma mensagem.

A message box pode armazenar um número fixo de mensagens.

```
CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE
```

## API

Criar uma nova fila de mensagens.

```
void cyg_mbox_create(cyg_handle_t* handle,cyg_mbox* mbox);
```

Eliminar uma fila de mensagens.

```
void cyg_mbox_delete(cyg_handle_t mbox);
```

Obter uma mensagem

```
void* cyg_mbox_get(cyg_handle_t mbox);
```

Obter uma mensagem com temporização

```
void* cyg_mbox_timed_get(cyg_handle_t mbox, cyg_tick_count_t abstime);
```

Tentar obter uma mensagem.

```
void* cyg_mbox_tryget(cyg_handle_t mbox);
```

Obter o numero de mensagens.

```
cyg_count32 cyg_mbox_peek(cyg_handle_t mbox);
```

Ver se há mensagens.

```
void* cyg_mbox_peek_item(cyg_handle_t mbox);
```

Colocar mensagem.

```
cyg_bool_t cyg_mbox_put(cyg_handle_t mbox, void* item);
```

Colocar mensagem com temporização

```
cyg_bool_t cyg_mbox_timed_put(cyg_handle_t mbox, void* item,
    cyg_tick_count_t abstime);
```

Tentar colocar mensagem com.

```
cyg_bool_t cyg_mbox_tryput(cyg_handle_t mbox,void* item);
```

Verificar se há alguma thread aguardando mensagem.

```
cyg_bool_t cyg_mbox_waiting_to_get(cyg_handle_t mbox);
```

Verificar se há alguma thread aguardando para colocar mensagem.

```
cyg_bool_t cyg_mbox_waiting_to_put(cyg_handle_t mbox);
```

## Exemplo

```
/*  Gestor de memória para blocos de dimensão fixa pré alocados
*/
#include <stdlib.h>
#include <stdio.h>
#include <cyg/kernel/kapi.h>
#include <cyg/infra/diag.h>

static cyg_mbox pool_mbox;
static cyg_handle_t pool_handle;

static void pool_create(cyg_handle_t * handle, cyg_mbox * mbox, int n, int
size) {
    int i;
    cyg_mbox_create(handle, mbox);
    for (i = 0; i < n; ++i) {
        void * p = malloc(size);
        cyg_mbox_put(*handle, p);
    }
}

static void pool_destroy(cyg_handle_t handle) {
    void * p = cyg_mbox_tryget(handle);
    while (p) {
        free(p);
        p = cyg_mbox_tryget(handle);
    }
    cyg_mbox_delete(handle);
}

static void * pool_alloc(cyg_handle_t handle) {
    return cyg_mbox_get(handle);
}

static void  pool_dealloc(cyg_handle_t handle, void * p) {
    cyg_mbox_put(handle, p);
}

static cyg_thread thread_desc[2];
static char thread_stack[2][4096];
static cyg_handle_t thread_hdl[2];

static cyg_mbox queue_mbox;
static cyg_handle_t queue_handle;
```

```
static void producer(cyg_addrword_t data) {
    int delay;
    while (1) {
        /*  Get memory to message */
        void * p = pool_alloc(pool_handle);
        /*  Produce some data */
        delay = 200 + (rand() % 50);
        cyg_thread_delay(delay);

        /*  Sending message */
        cyg_mbox_put(queue_handle, p);
        diag_printf("Producer\n");
    }
}

static void consumer(cyg_addrword_t data) {
    int delay;
    while (1) {
        /*  Get message */
        void * p = cyg_mbox_get(queue_handle);
        /*  Consuming the data */
        delay = 200 + (rand() % 50);
        cyg_thread_delay(delay);
        /*  Freeing memory */
        pool_dealloc(pool_handle, p);
        diag_printf("Consumer\n");
    }
}

void cyg_user_start(void) {
    printf( "eCos message box example\n\n" );
    pool_create(&pool_handle, &pool_mbox, 10, 100);
    cyg_mbox_create(&queue_handle, &queue_mbox);
    cyg_thread_create(4, producer, (cyg_addrword_t) 0,
        "Thread consumer", (void *) thread_stack[0], sizeof(thread_stack[0]),
        &thread_hdl[0], &thread_desc[0]);
    cyg_thread_create(4, consumer, (cyg_addrword_t) 1,
        "Thread producer", (void *) thread_stack[1], sizeof(thread_stack[0]),
        &thread_hdl[1], &thread_desc[1]);
    cyg_thread_resume(thread_hdl[0]);
    cyg_thread_resume(thread_hdl[1]);
}
```

## *Interrupt*

O eCos usa um esquema de processamento das interrupções em duas etapas: Interrupt Service Routine (ISR) e Defered Service Routine (DSR).

O objectivo é a redução da latência à interrupção.

A rotina ISR é executada com as interrupções inibidas e a sua duração deve ser a mínima possível.

O restante processamento é executado na DSR com as interrupções desinibidas (não comprometendo a latência).

A comutação de *threads* pode ser suspensa pelo núcleo ou pela aplicação o que não permite a execução de DSRs.

As ISR's têm prioridades de execução sobre as DSR's e as DSR's têm prioridade de execução sobre as *threads*.

No caso em que o atendimento do periférico ser feito na DSR, e para evitar que a ISR reentre, a ISR mascara a interrupção e só a respectiva DSR voltará a desmascarar.

A DSR possui um parâmetro que indica o número de activações da ISR desde a última execução da DSR. Um valor superior a 1 significa reentrada. No caso de o atendimento do periférico ser feito na ISR, um valor superior a 1 significa sistema sobrecarregado.

As ISR's não podem usar funções de sincronização relacionadas com a comutação de tarefas uma vez que esta está inibida durante o processamento da ISR.

A ISR sinaliza a respectiva DSR através do valor de retorno da função.

```
cyg_uint32 isr( cyg_vector_t vector, cyg_addrword_t data) {
    . . .
    return( CYG_ISR_HANDLED | CYG_ISR_CALL_DSR );
}
```

Se a comutação de *threads* não estiver bloqueada a DSR executa imediatamente.

A DSR não pode invocar uma função de sincronização que implique um bloqueio.

A DSR apenas pode usar funções de sincronização que não impliquem bloqueio para sinalizar as threads.

Podem ser associados vários objectos de interrupção numa mesma entrada de interrupção para permitir a sua partilha por vários dispositivos.

O processamento de uma interrupção pode implicar a chamada sucessiva de várias ISRs. Esta sucessão de chamadas termina quando uma ISR devolver a sinalização CYG_ISR_HANDLED.

## API

Criar um objecto para tratar uma interrupção.

```
void cyg_interrupt_create(cyg_vector_t vector, cyg_priority_t priority,
cyg_addrword_t data, cyg_ISR_t * isr, cyg_DSR_t * dsr,
cyg_handle_t * handle, cyg_interrupt * desc);
```

Eliminar um objecto de interrupção.

```
void cyg_interrupt_delete(cyg_handle_t interrupt);
```

Activar o objecto de interrupção. Necessário quando se partilha o vector de interrupção.

```
void cyg_interrupt_attach(cyg_handle_t interrupt);
```

Desactivar o objecto de interrupção.

```
void cyg_interrupt_detach(cyg_handle_t interrupt);
```

Configurar a sensibilidade da entrada de interrupção.

```
void cyg_interrupt_configure
    (cyg_vector_t vector, cyg_bool_t level, cyg_bool_t up);
```

Sinalizar ao controlador de interrupções o fim do atendimento da interrupção.

```
void cyg_interrupt_acknowledge(cyg_vector_t vector);
```

Inibir as interrupções globalmente.

```
void cyg_interrupt_disable(void);
```

Desinibir as interrupções globalmente.

```
void cyg_interrupt_enable(void);
```

Mascarar a entrada de interrupção indicada.

```
void cyg_interrupt_mask(cyg_vector_t vector);
```

Mascarar a entrada de interrupção indicada (sem protecção).

```
void cyg_interrupt_mask_intunsafe(cyg_vector_t vector);
```

Desmascarar a entrada de interrupção indicada.

```
void cyg_interrupt_unmask(cyg_vector_t vector);
```

Desmascarar a entrada de interrupção indicada (sem protecção).

```
void cyg_interrupt_unmask_intunsafe(cyg_vector_t vector);
```

## Exemplo

Situação típica de atendimento de uma interrupção.

```
cyg_uint32 isr( cyg_vector_t vector, cyg_addrword_t data) {
    cyg_interrupt_mask(vector);
    cyg_interrupt_acknowledge(vector);
    . . .
    return( CYG_ISR_HANDLED | CYG_ISR_CALL_DSR );
}
```

Mascarar a entrada de interrupção.

Sinalizar o controlador de interrupções do atendimento da interrupção.

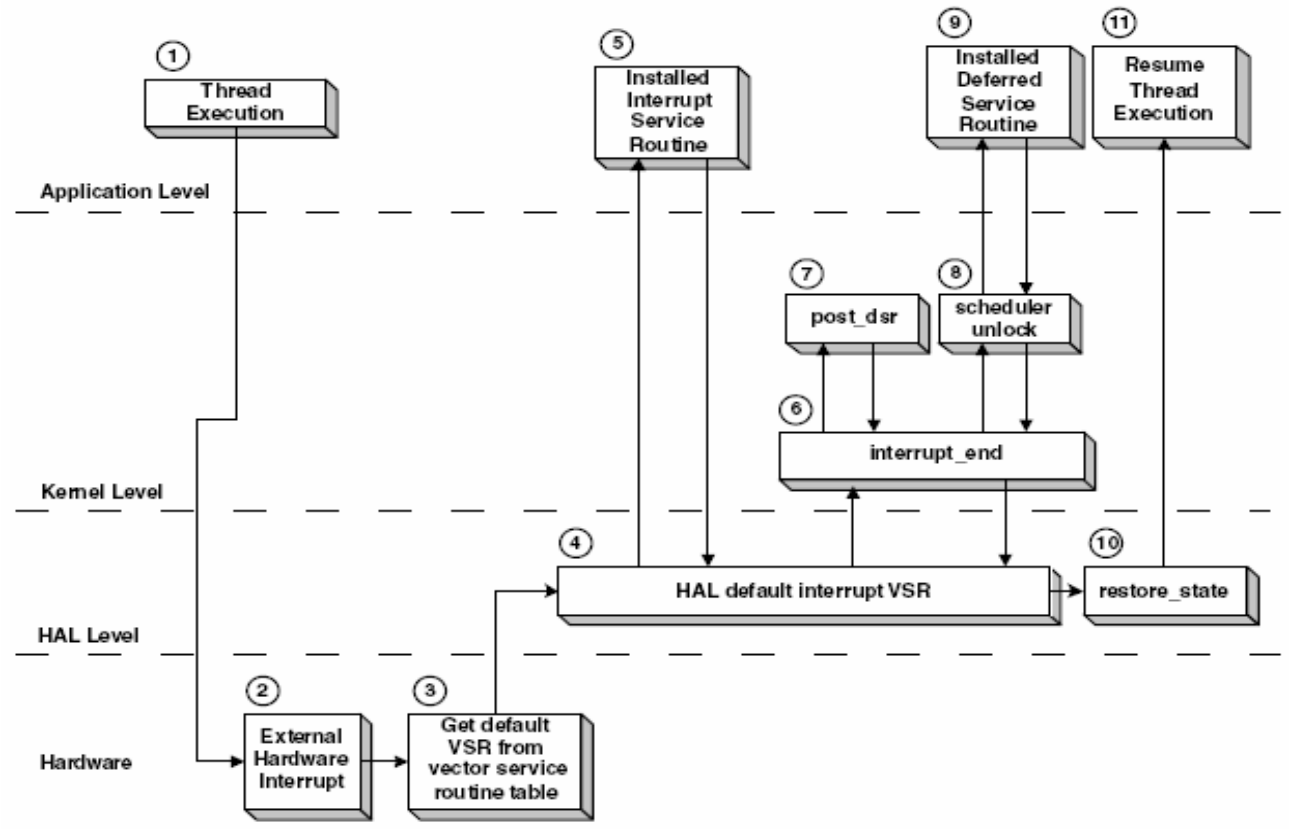
```
void dsr(
    cyg_vector_t vector, cyg_ucount32 count, cyg_addrword_t data) {

    . . .

    cyg_semaphore_post(&ready);
    cyg_interrupt_unmask(vector);
}
```

Sinalizar a *thread*.

Desmascarar a entrada de interrupção.

```
#include <cyg/infra/diag.h>
#include <cyg/kernel/kapi.h>
#include <cyg/hal/hal_io.h>

static cyg_interrupt intr_desc;
static cyg_handle_t intr_handle;
static cyg_sem_t ready;

#define CYGNUM_HAL_PRI_HIGH         0

static cyg_uint32 isr(cyg_vector_t vector, cyg_addrword_t data) {
    cyg_interrupt_mask(vector);
    cyg_interrupt_acknowledge(vector);
    return (CYG_ISR_HANDLED | CYG_ISR_CALL_DSR);
}

static void dsr(cyg_vector_t vector, cyg_ucount32 count, cyg_addrword_t data) {
    HAL_WRITE_UINT32(CYGARC_HAL_LPC2XXX_REG_SCB_BASE +
CYGARC_HAL_LPC2XXX_REG_EXTINT,
        CYGARC_HAL_LPC2XXX_REG_EXTxxx_INT0);
    cyg_semaphore_post(&ready);
    cyg_interrupt_unmask(vector);
}
```

```
#define THREAD_STACK_SIZE  (8192 / sizeof(int))
static int thread_stack[THREAD_STACK_SIZE];

static cyg_handle_t thread_handle;
static cyg_thread thread_desc;

void thread(cyg_addrword_t index) {
    cyg_uint32 count = 0;
    /* Conectar sinal EINT0 ao pino do circuito integrado */
    HAL_WRITE_UINT32(CYGARC_HAL_LPC2XXX_REG_PIN_BASE +
CYGARC_HAL_LPC2XXX_REG_PINSEL1, 1);
    cyg_interrupt_unmask(CYGNUM_HAL_INTERRUPT_EINT0);
    while (1) {
        cyg_semaphore_wait(&ready);
        diag_printf("Interrupt %d\n", count++);
    }
}

void cyg_user_start(void) {
    diag_printf("Object Interrupt test\n");

    cyg_semaphore_init(&ready, 0);

    cyg_thread_create(12, thread, 0, "Interrupt test thread",
        &thread_stack, THREAD_STACK_SIZE,
        &thread_handle, &thread_desc);

    cyg_thread_resume(thread_handle);

    cyg_interrupt_create(CYGNUM_HAL_INTERRUPT_EINT0, CYGNUM_HAL_PRI_HIGH, 0,
        isr, dsr, &intr_handle, &intr_desc);

    cyg_interrupt_attach(intr_handle);
    cyg_interrupt_configure(CYGNUM_HAL_INTERRUPT_EINT0, true, true);
    cyg_interrupt_acknowledge(CYGNUM_HAL_INTERRUPT_EINT0);
}
```



# Temporizações

As temporizações são necessárias em diferentes níveis de um sistema: escalonamento em tempo repartido; refrescamento de memória dinâmica por software; temporizações em protocolos de comunicações; programar actividades no tempo; etc

**hard-timer** – boa precisão; timer de hardware activa rotina de interrupção que executa todo o trabalho; cria sobrecarga de processamento; microsegundo e nanosegundos.

**soft-timer** – pouca precisão; baseados num único timer de hardware; o trabalho não é feito na rotina de interrupção, é feito posteriormente por uma *thread* própria para o efeito; minimiza a sobrecarga de processamento no caso de ser muito utilizado; milisegundos e segundos.

**Real-Time Clock (hora-minuto-segundo-dia-mês-ano)**

É suportado por unidade autónoma com bateria para manter o funcionamento mesmo depois de desligado o sistema.

**System Clock**

Trata-se de uma entidade em memória, normalmente um contador que é incrementado numa rotina de interrupção activada por um *timer* de hardware. Os tempos são relativos ao momento de arranque do sistema.

## Meios de temporização do eCos

**Counter** – serve para contar. Podem-se-lhe associar alarmes que permitem a chamada de uma função quando o contador atinge um certo valor. O contador é incrementado explicitamente pela função **`cyg_counter_tick`**. Chamada normalmente no contexto de uma DSR consequência de um evento externo.

**Clock** – são contadores associados a hardware específico que gera tiques a intervalos regulares (hardware timer). O núcleo já dispõe de um objecto destes cujo *handle* pode ser obtido com a

função **`cyg_real_time_clock`**. Mais objectos destes podem ser criados ou eliminados. O objecto **clock** não é automaticamente associado a hardware. É necessário que separadamente o hardware gerador de eventos seja inicializado, e que a função **`cyg_counter_tick`** seja chamada da respectiva DSR.

**Alarm** – é um meio de gerar eventos baseando-se no valor de um **counter**.

## Counter API

Criar um novo contador.

```
void cyg_counter_create( cyg_handle_t *counter, cyg_counter *the_counter);
```

Eliminar um contador.

```
void cyg_counter_delete(cyg_handle_t counter);
```

Incrementa o contador de uma unidade.

```
void cyg_counter_tick(cyg_handle_t counter);
```

Obter o valor do contador.

```
cyg_tick_count_t cyg_counter_current_value(cyg_handle_t counter);
```

Definir o novo valor do contador.

```
void cyg_counter_set_value(
     cyg_handle_t counter, cyg_tick_count_t new_value);
```

## Clock API

Criar um novo relógio.

```
void cyg_clock_create(
     cyg_resolution_t resolution,cyg_handle_t *handle, cyg_clock *clock );
```

Eliminar um relógio.

```
void cyg_clock_delete(cyg_handle_t clock);
```

Transforma um relógio num contador.

```
void cyg_clock_to_counter(cyg_handle_t clock, cyg_handle_t *counter);
```

Alterar a resolução de um relógio. Resolução: nanosegundos por tique.

```
void cyg_clock_set_resolution(
     cyg_handle_t clock,cyg_resolution_t resolution );
```

Obter a resolução de um relógio.

```
cyg_resolution_t cyg_clock_get_resolution(cyg_handle_t clock);
```

Obter o *handle* do relógio do sistema.

```
cyg_handle_t cyg_real_time_clock();
```

Obter o valor em tiques do relógio do sistema.

```
cyg_tick_count_t cyg_current_time();
```

A **`struct cyg_resolution`** representa a resolução de um **clock** em nanosegundos por tique.

```
struct cyg_resolution {
        cyg_uint32  dividend;
        cyg_uint32  divisor;
};
```

Se a resolução for de 10000000 ns por tique, a que corresponde uma frequência de 100 Hz,

os valores para dividendo e divisor seriam respectivamente 1000000000 e 100.

Para converter tiques para nanosegundos:

```
nanoseconds = ticks * dividend / divisor
```

Para converter tiques para segundos:

```
seconds = ticks / ((divisor * 1000000000LL) / dividend)
```

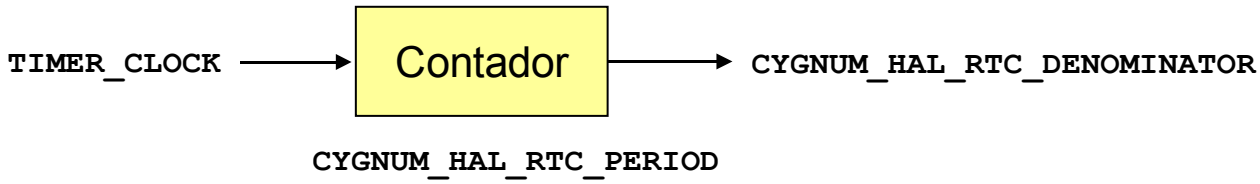
As seguintes constantes definem a resolução do **clock** de sistema. Para efectuar conversões entre tiques e segundos é preferível usarem-se estas constantes, pois as operações serão resolvidas em tempo de compilação.

**CYGNUM_HAL_RTC_DENOMINATOR** (divisor) – tiques por segundo.

**CYGNUM_HAL_RTC_NUMERATOR** (dividend) – nanosegundos por segundo.

**CYGNUM_HAL_RTC_NUMERATOR / CYGNUM_HAL_RTC_DENOMINATOR** (resolution) – nanosegundos por tique.

Estas constantes são definidas na configuração do eCos e são usadas para programar o hardware de modo a que seja gerada uma interrupção periódica com a frequência pretendida.




```
CYGNUM_HAL_RTC_PERIOD = TIMER_CLOCK / CYGNUM_HAL_RTC_DENOMINATOR
```

## Alarm API

Criar um novo alarme.

```
void cyg_alarm_create(cyg_handle_t counter,
     cyg_alarm_t * alarm_function, cyg_addrword_t data,
     cyg_handle_t * handle, cyg_alarm * desc);
```

Eliminar um alarme.

```
void cyg_alarm_delete(cyg_handle_t alarm);
```

Preparar dados do alarme.

```
void cyg_alarm_initialize(cyg_handle_t alarm,
     cyg_tick_count_t trigger, cyg_tick_count_t interval);
```

Armar o alarme.

```
void cyg_alarm_enable(cyg_handle_t alarm);
```

Desarmar o alarme.

```
void cyg_alarm_disable(cyg_handle_t alarm);
```

## Exemplo 1

```
#include <cyg/kernel/kapi.h>

static cyg_counter counter_obj;
static cyg_handle_t counter_hdl;
static cyg_alarm alarm_obj;
static cyg_handle_t alarm_hdl;

static unsigned long index = 0;
```

```
void counter_thread() {
    while (1) {
         cyg_thread_delay(10);
         cyg_counter_tick(counter_hdl);
    }
}
void alarm_handler(cyg_handle_t alarm_handle, cyg_addrword_t data) {
    *(unsigned long *)data++;
}

void cyg_user_start(void)  {
    cyg_counter_create(&counter_hdl, &counter_obj);
    cyg_alarm_create(counter_hdl, alarm_handler,
         (cyg_addrword_t)&index, &alarm_hdl, &alarm_obj);
    cyg_alarm_initialize(alarm_hdl, 12, 6);
}
```

## Exemplo 2

```
#include <cyg/kernel/kapi.h>

static cyg_handle_t counter_hdl;
static cyg_handle_t sys_clk;
static cyg_handle_t alarm_hdl;
static cyg_tick_count_t ticks;
static cyg_alarm_t alarm_handler;
static cyg_alarm alarm_obj;

unsigned long index;

void cyg_user_start( void ) {
    sys_clk = cyg_real_time_clock();
    cyg_clock_to_counter(sys_clk, &counter_hdl);

    cyg_alarm_create(counter_hdl, alarm_handler, (cyg_addrword_t)&index,
              &alarm_hdl, &alarm_obj);

    cyg_alarm_initialize( alarm_hdl, cyg_current_time() + 100, 100 );
}

void alarm_handler( cyg_handle_t alarm_handle, cyg_addrword_t data ) {
    *((unsigned long *)data)++;
}
```

## Temporizações de precisão

No eCos há dois recursos que permitem efectuar temporizações de maior precisão que o tique que é normalmente na ordem dos 10 ms.

**`HAL_CLOCK_READ(value)`** – lê o valor do contador divisor de frequência usado para gerar a interrupção de relógio do sistema.

**`HAL_DELAY_US(us)`** – usa expressamente um *timer* de hardware para efectuar a temporização.

```
#include <cyg/hal/hal_intr.h>

#define timer_start(x)  hal_clock_read(&x)

#define timer_restart(i, t) ({ \
    i += t;  \
```

```
    i = (i > CYGNUM_HAL_RTC_PERIOD) ? i - CYGNUM_HAL_RTC_PERIOD : i; \
})

#define timer_elapsed(i, e) ({ \
    cyg_uint32 n;     \
    hal_clock_read(&n);    \
    e = (n < i) ? CYGNUM_HAL_RTC_PERIOD - i + n : (n - i);     \
})
```

# Referências


1. Manuais no sitio do eCos: http://ecos.sourceware.org/.

2. Embedded Software Development with eCos, Anthony J. Massa. Este livro está disponível com licença GNU em muitos sítios da Internet.