INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA
SERVIÇO DE DOCUMENTAÇÃO E PUBLICAÇÕES

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA
Área Departamental de Engenharia de Eletrónica e Telecomunicações e de Computadores

Projeto de uma Câmara em Rede em FPGA

PEDRO GUILHERME AMARAL DA PONTE
(Bacharel)

Trabalho Final de Mestrado para Obtenção do Grau de Mestre em Engenharia de Eletrónica e Telecomunicações

Orientador:
Prof.º Doutor Mário P. Véstias

Júri:
Presidente: Prof.º Fernando Manuel Ascenso Fortes
Vogais:
Prof.º Horácio Cláudio de Campos Neto
Prof.º Doutor Mário P. Véstias

Dezembro de 2013

Agradecimentos

Gostaria de agradecer, em primeiro lugar, à minha família pelo apoio incondicional dado e pela compreensão demonstrada face à minha ausência em alguns momentos.
Agradeço a todos os professores do ISEL que me acompanharam durante o meu percurso académico transferindo o seu conhecimento e que, seguramente têm uma grande influência na pessoa que eu sou a nível profissional.
Ao professor Dr. Mário Véstias, que na qualidade de orientador deste trabalho de projeto, demonstrou sempre uma enorme disponibilidade para a orientação do mesmo.

Resumo e Palavras-chave

O recurso a *FPGA* para o desenvolvimento de um sistema embebido, como por exemplo o de uma Câmara de Rede, tem sido largamente adotado devido à possibilidade de síntese dos diversos componentes do sistema num único circuito impresso. Desta forma, conseguem-se sistemas com grande complexidade, desempenho e com *hardware* especializado, para implementar aceleramento, sem prejuízo da área ocupada.

Este projeto consiste em integrar a câmara de vídeo *TRDB_D5M* [4] num sistema baseado no processador *NIOS*[1] sintetizado em *FPGA*. O sistema será constituído pelos componentes necessários à aquisição de imagem, conversão para *RGB*, compressão para *JPEG* e interface para com o utilizador.

A interface para com o utilizador será feita via *Ethernet* através de *Web Browser* onde o utilizador poderá verificar as características do sistema, monitorizar as estatísticas referentes às tramas *Ethernet*, efetuar algumas configurações e visualizar imagem.

Palavras-chave: *FPGA, NIOS, RGB, JPEG, TRDB_D5M*, Câmara em Rede, Sistema Embebido.

Abstract and Keywords

The use of FPGA's for embedded systems development, such as a Web Cam, has been widely adopted due to the possibility of synthesis of the various system components on a single printed circuit. Thus can be systems with great complexity, performance and specialized hardware to implement acceleration, without prejudice to the occupied area.

This project is to integrate the camcorder *TRDB_D5M* [4] into a *NIOS* [1] processor-based system [1] synthesized in a FPGA. The system will consist of the necessary components for image acquisition, conversion to RGB, JPEG compression, and interface with the user.

The interface with the user is done via Ethernet through Web Browser where the user can check the system characteristics, monitor Ethernet frames statistics, make some settings and view image.

Keywords: *FPGA, NIOS, RGB, JPEG, TRDB_D5M*, Network Camera, Embedded System.

Índice

Lista de Figuras

Lista de Código

Acrónimos

FPGA	Field Programmable Gate Array
RGB	Sistema de cores aditivas formado por Vermelho (Red), Verde (Green) e Azul (Blue)
JPEG	Joint Photographic Experts Group
ISR	Interrupt Service Routine
MJPEG	Motion JPEG
H.264	Técnica de compressão baseada no MPEG-4 Part 10 ou AVC (Advanced Video Coding)
MPEG	Moving Picture Experts Group
IP	Internet Protocol
HTTP	Hypertext Transfer Protocol
SSL	Secure Sockets Layer
TLS	Transport Layer Security
QoS	Quality of service
FTP	File Transfer Protocol
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
DNS	Domain Name System
DynDNS	Dynamic DNS
NTP	Network Time Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
IGMP	Internet Group Management Protocol
ICMP	Internet Control Message Protocol
RTCP	RTP Control Protocol
ARP	Address Resolution Protocol
DHCP	Dynamic Host Configuration Protocol
RTSP	Real Time Streaming Protocol
ALU	Unidade lógica e Aritmética
ISR	Interrupt Service Routine
FIFO	First In, First Out
GPIO	General Purpose Input/Output
API	Aplication Protocol Interface

Capítulo 1 - Introdução

Atualmente, os sistemas embebidos, são utilizados para desempenhar inúmeras funcionalidades devido ao facto de estes normalmente apresentarem características mais apetecíveis ao consumidor, como por exemplo a dimensão, peso e preço.

A sua empregabilidade é muito vasta, desde um telemóvel até a um sistema de automatização industrial.

O projeto de uma Câmara em Rede é, também, um exemplo onde um sistema embebido pode ser utilizado tirando partido do fato de este ser sintetizado numa *FPGA* para incluir nesta os módulos necessários, sem a necessidade de recorrer a *hardware* externo.

1.1 – Estado da Arte

As Câmaras de Rede são, nos dias de hoje, vulgarmente utilizadas, em prejuízo das câmaras convencionais analógicas, visto que podem ser ligadas às redes *Ethernet* normalmente utilizadas sem a necessidade de uma rede dedicada de cabo coaxial para o efeito. Estas, para além de fornecerem imagem, podem fornecer ao utilizador uma série de aplicações tais como deteção de movimento, tratamento da imagem, visualização remota, entre outras.

A banalização do acesso à Internet e o aumento da largura de banda oferecida neste serviço fez com que fosse possível a utilização destes dispositivos para uma série de aplicações.

O aumento da criminalidade tem, também, impulsionado a procura destes dispositivos dando origem a uma maior competitividade, por parte dos fabricantes e levando-os a desenvolver soluções cada vez mais atrativas e menos dispendiosas.

No que diz respeito às especificações técnicas, quase todas baseiam- se em sistemas embebidos, com interface *Ethernet*, e apresentam resoluções na ordem dos megapixéis com mecanismos de compressão *MJPEG*, *MPEG-4* e *H.264*.

1.1.1 – Soluções Comerciais

O mercado, atualmente, disponibiliza uma enorme oferta de soluções tanto ao nível de especificações técnica como em função da aplicabilidade. Desde Câmaras fixas, móveis, panorâmicas, para interior e exterior.

De seguida podemos verificar algumas soluções disponibilizadas pelo fabricante AXIS.

AXIS M10

A gama de Câmaras M10 da AXIS [27] está inserida na categoria de Câmaras fixas e disponibiliza uma série de Câmaras em função das características técnicas pretendidas.

Figura 1 - Câmara AXIS Série M10 (Fonte: [27])

Nesta gama estão disponíveis Câmaras com diferentes resoluções, conectividade opcional a redes *WI-FI* (IEEE 802.11), suporte para áudio, entre outras características.

Em ambos os casos, a conectividade *Ethernet* está disponível e um ritmo de compressão de 30 *fps* (Tramas por segundo) para os formatos H.264, MJPEG e MPEG-4.

AXIS M11

A gama de Câmaras M11 da AXIS [27] está mais vocacionada para soluções em ambientes exteriores sendo idênticas à série anterior (M10) mas, no entanto, apresentam uma proteção por forma a estar imune às intempéries.

Figura 2 - Câmara AXIS Série M11 (Fonte: [27])

AXIS Q60

A gama de Câmaras Q60 da AXIS [27] são *Speed Domes*, para interior e exterior, normalmente utilizadas em soluções de caracter mais profissional.
Este tipo de Câmaras são concebidas para operar em ambientes mais extremos justificando a presença de ventoinhas permitindo que estas possam operar em ambientes com temperaturas entre os -20 e +75 ºC.
As Câmaras *Speed Domes* têm a particularidade de serem movíveis, normalmente através do envio de comandos, e através de caminhos pré programados podem efetuar varrimentos nos mesmos.

Figura 3 - Câmara AXIS Série Q60 (Fonte: [27])

Em ambos os casos a AXIS disponibiliza uma API que possibilita uma integração das Câmaras numa aplicação.
De Seguida podemos visualizar algumas características genéricas das Câmaras AXIS.

Memória	64/128 MB RAM e 32/128 MB FLASH
Protocolos de rede	IPv4/v6, HTTP, HTTPSb, SSL/TLS, QoS Layer 3 DiffServ, FTP, SMTP,UPnP, SNMP, DNS, DynDNS, NTP, RTSP, RTP, TCP, UDP, IGMP, RTCP, ICMP, DHCP, ARP, SOCKS
Configuração de Imagem	Compressão, Cor, Brilho, Contraste, Mascara de Privacidade.
Sensor	Progressive scan RGB CMOS
Resolução	640x480 to 160x120

1.2 - Objetivos do Trabalho

Este Trabalho de Projeto pretende desenvolver uma Câmara em Rede a partir da síntese de um processador *NIOS* [1], e respetivos periféricos, numa *FPGA* por forma a disponibilizar uma arquitetura com os recursos necessários para posterior desenvolvimento aplicacional.

Para o efeito iremos recorrer à placa de desenvolvimento *DE-115* [4], da Altera, e à câmara *TRB-D5M* [22] que liga à placa através da interface *GPIO* desta placa, conforme nos sugere a figura que se segue.

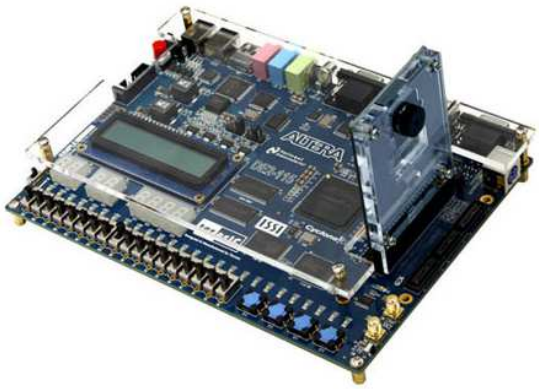

Figura 4 - Placa DE-115 com Câmara D5M (Fonte: [4])

O core do processador *NIOS* utiliza interfaces *Avalon-MM* (*Avalon Memory Mapped*) [16] para comunicar com os periféricos, entre os quais os dispositivos de memória, através do *System Interconnect Fabric* que tem a responsabilidade de garantir a exclusão mútua no acesso ao *bus* do sistema.

Na figura que se segue, onde consta a arquitetura que se pretende implementar, podemos verificar que esta será constituída pelo processador *NIOS*, dois dispositivos de memória *SRAM* e *SDRAM*, ligado aos restantes periféricos através do *System Interconnect Fabric*.

Para implementar a conectividade *Ethernet* o sistema recorre ao módulo *TSE*, disponibilizado pela *Altera*, que utiliza módulos *SG-DMA* para transferir as tramas entre a memória do sistema e o módulo. Os módulos *SG-DMA* processam descritores que têm os atributos necessários para descrever as zonas de memória onde residem as tramas *Ethernet*, garantir exclusão mútua às mesmas e criar listas.

Com o intuito de integrar a Câmara *D5M* [22] no sistema irá ser implementado, de raiz, o módulo *BAYER* especializado para captar a imagem, converter no formato *RGB* e enviar a trama para a memória *SRAM* do sistema. O módulo *I2C* será adicionado para que o sistema, a partir do *device driver*, possa configurar os registos internos da Câmara *D5M*.

O módulo *ENCJPEG*, que corresponde a uma adaptação do módulo *EV_JPEG_ENC* [24] para o sistema aqui desenvolvido, irá assegurar a compressão das tramas *RGB* para *JPEG* para posterior encapsulamento em *MJPEG* (*Motion JPEG*) [25] por parte da aplicação.

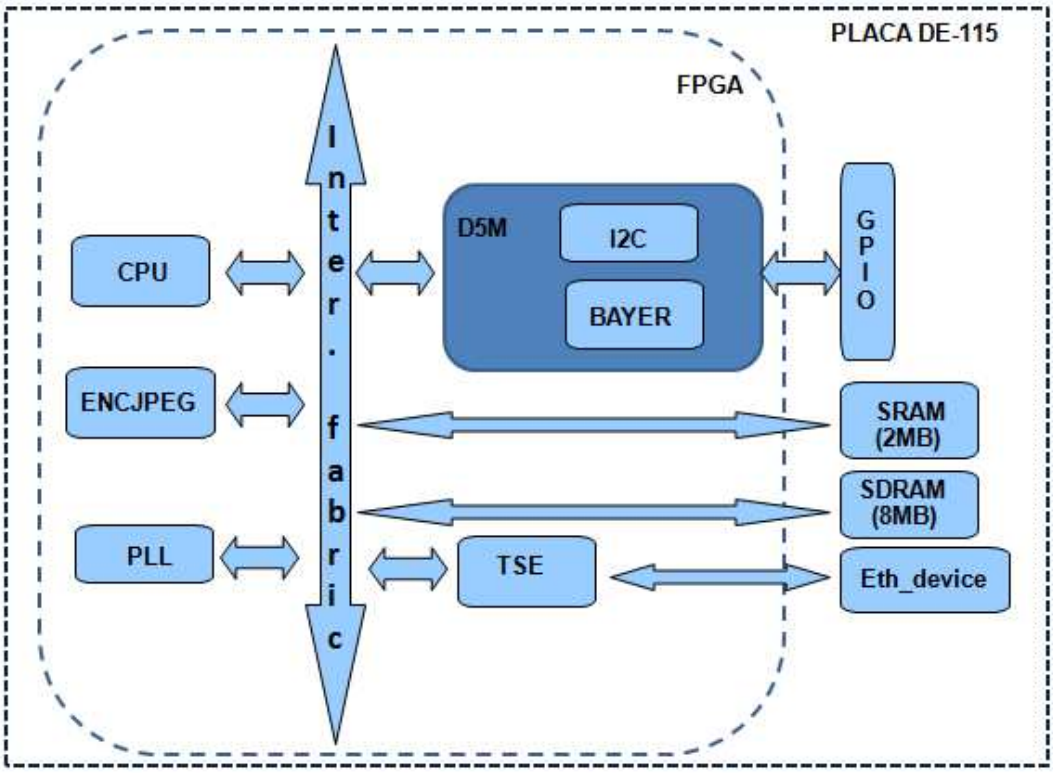

Figura 5 - Arquitetura do sistema

Serão alvo de estudo os diversos módulos constituintes do sistema, nomeadamente na forma como são instanciados e desenvolvimento do respetivo *device driver* por forma a disponibilizar uma *API* para posterior desenvolvimento aplicacional.

O Capítulo 4, que aborda o desenvolvimento do módulo de aquisição de imagem, demonstra a forma como se desenvolve um módulo, em *VHDL*, por forma a ligar ao sistema através de interfaces *Avalon* [16].

Outro objetivo deste Trabalho de Projeto consiste em estudar a forma como se recorre a *Software Open Source* e adaptá-lo para a arquitetura do sistema (*porting*). Neste contexto, para implementação do *stack TCP/IP* [17] será utilizado o pacote *uIP* [18].

1.3 - Organização do Projeto

Tendo em conta de que este trabalho de Projeto consiste no desenvolvimento de uma Câmara em Rede baseada num sistema embebido, este irá ser desenvolvido de uma forma faseada, partindo-se de um simples sistema constituído pelo processador até atingir um sistema com todos os periféricos necessários à implementação do sistema pretendido.

O Capítulo 2 aborda a implementação do sistema embebido, ainda sem a inclusão da Câmara *D5M* sendo os componentes, de que fazem parte integrante deste sistema, estudados em tópicos distintos deste Capítulo. Nomeadamente o Controlador de Interrupções interno do processador *NIOS*, acompanhado de uma aplicação com recurso a um dos botões da placa de desenvolvimento *DE2_115* [4] para gerar um pedido de interrupção que irá efetuar *toggle* a um *led* desta placa, um *timer* para posterior implementação do relógio de sistema e o módulo de memória externa *SDRAM*.
O Capítulo 3 adicionam ao sistema o módulo *Ethernet*, recorrendo à biblioteca disponibilizada pela *Altera*, desenvolvimento do respetivo *device driver*, *TCP/IP stack* e interface *socket*.
O Capítulo 4 explora toda a interatividade entre o sistema *NIOS* e a Câmara *D5M*, desde a captação dos pixéis, formatação para *RGB* e controlo da Câmara através do seu *device driver*.
O Capítulo 5 aborda o desenvolvimento aplicacional por forma a controlar os componentes do sistema e gerir a interface com o utilizador, através do desenvolvimento de uma aplicação servidora *http*.
Finalmente, no Capítulo 6, são apresentados os testes efetuados para determinar as performances do sistema desenvolvido bem como as características do sistema concebido após inclusão de todos os módulos.
Os módulos constituintes do sistema serão estudados em Capítulos, e tópicos, distintos sendo os pontos comuns considerados como implícitos. A título de exemplo, a forma como se integra um *device driver* na camada de abstração (*HAL- Hardware Abstraction Layer*) [8], disponibilizada pelo *BSP* [2] (*Board Support Package*), apenas é focada aquando o estudo do módulo *Ethernet*.

Capítulo 2 – Implementação do Sistema Nios

A implementação da Câmara em Rede passará pela síntese de um *SoC* (System on Chip), na *FPGA* da placa *DE-115* [4], constituído por um processador *NIOS* [1] e pelos diversos componentes periféricos por forma a implementar um sistema embebido.

O processador *NIOS* [1] é uma arquitetura a 32 *bits* e está disponível, para instanciação, em três versões distintas (Nios II/f, Nios II/s e Nios II/e) sendo a versão Nios II/s adotada neste Trabalho de Projeto, visto que a versão mais rápida Nios II/f não ser gratuita.

O core Nios II/s utiliza cerca de menos 20% da área na FPGA, em relação ao core Nios II/f, mas apresenta um desempenho inferior na ordem dos 40%.

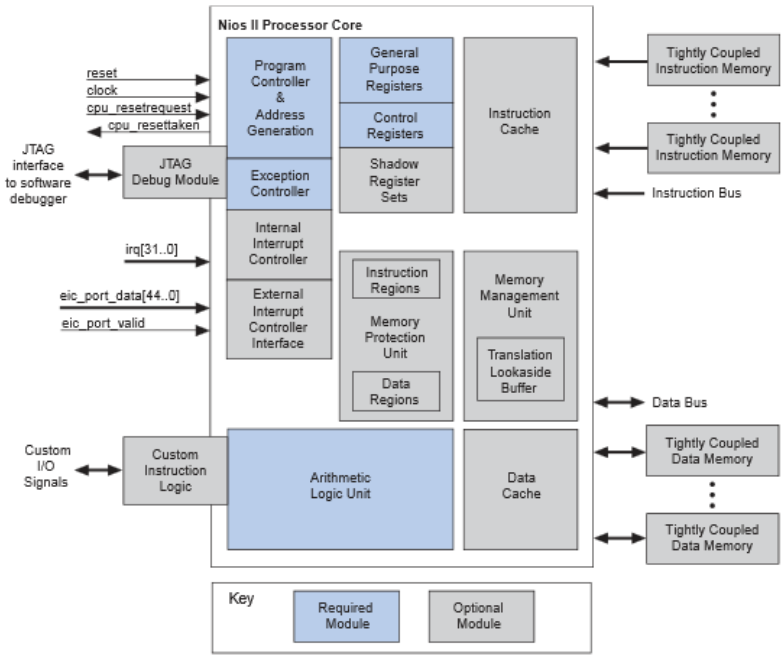

Figura 6 - Arquitetura do Processador Nios (Fonte: [1])

Na figura que se antecede podemos verificar a arquitetura genérica do Processador Nios [1], sendo que o core aqui utilizado não dispõe de Cache de dados. Do *bus* de endereço a 32 *bits* apenas são utilizados os 31 *bits LSB* permitindo um endereçamento de 2GB.

Apesar de o Processador apresentar um *bus* para instruções, é possível implementar um sistema sem recurso a este *bus* visto que este dispõe de *cache* para instruções.

No que diz respeito a interrupções, podemos optar pelo controlador interno, que suporta até 32 entradas de interrupção **irq[31…0]**, ou por um controlador externo recorrendo, para o efeito, à entrada **eic_port**.

A *ALU* implementa as operações adição, subtração, multiplicação, divisão e *shift* bem

como a implementação de operações lógicas mais complexas para implementar aceleramento, normalmente referido como C2H *hardware acceleration* [28]. Esta técnica consiste em implementar *hardware* que implementa uma função lógica que pode, ou não, aceder diretamente à *ALU* do processador com vista a minimizar os tempos de acesso.

O processador Nios II contém 32 registos internos de controlo e, opcionalmente, 63 bancos adicionais que podem ser utilizados para salvaguardar o contexto de execução entre sucessivos pedidos de interrupção.

2.1 - Exceções e Interrupções

O processador *Nios II* dispõe de um controlador de exceções que permite dar ao programador o controlo sobre as exceções que ocorrem no sistema. Existem diversas situações que podem originar exceções como, por exemplo, utilização de *opcodes* não reconhecidos pelo processador e pedidos de interrupções de dispositivos externos ao processador.

Uma exceção causa, de forma atómica, a salvaguarda do contexto atual do processo e passagem do *program counter* [1] **(PC)** para o respetivo endereço configurado no vetor de exceções. Depois de atendida a rotina de interrupção, é da responsabilidade da rotina *ISR* (*Interrupt Service Routine*) devolver o controlo do processador à função interrompida bem como passar para os registos o contexto anteriormente guardado.

Para atendimento de interrupções a dispositivos externos, o processador *Nios II* disponibiliza até 32 entradas que são configuradas por software tanto ao nível da ativação e desativação como ao nível da prioridade.

A ativação do controlador de interrupções é feita ativando o *bit* **PIE** do registo **STATUS** [1] enquanto a ativação individual de cada uma das entradas de interrupção é feita ativando cada um dos *bits* do registo **IENABLE** [1].

O controlador de interrupções interno do Nios *II* não é vetorizado, ou seja, quando o processador atende uma interrupção e salta para o endereço configurado no vetor de exceções, o programador terá de efetuar mais testes para verificar qual o dispositivo que pediu a interrupção. Desta forma terá de se definir uma rotina, normalmente designada por *ISR (Interrupt Service Routine)* para então nesta efetuar a de multiplexagem para a rotina que implementa o atendimento ao periférico em causa.
Uma forma mais eficiente de se efetuar atendimento a pedidos de interrupção consiste em recorrer à implementação vetorizada, que não faz parte integrante do processador Nios II.
Na implementação vetorizada, quando é efetuado um pedido de interrupção, o controlador disponibiliza de imediato o *entry point* da rotina de atendimento ao periférico.

2.1.1 - O Controlador de Interrupções Interno (IIC)

Conforme referido na introdução deste tópico, as interrupções fazem parte de uma das exceções possíveis que podem ocorrer durante o normal funcionamento de um sistema e como tal, para as implementarmos teremos de em primeiro implementar a

resposta do sistema às diversas exceções.
A resposta de um sistema implementado com um processador *Nios II* a uma exceção é a seguinte:

- Guarda o conteúdo do registo de estado **STATUS** para o registo **ESTATUS** [1].
- Limpa (coloca a '0') a *flag* **U** do registo de estado **STATUS** colocando o processador a operar no modo **supervisor**. Nos sistemas sem *MMU (Memory Management Unit)* esta *flag* está sempre a '0'.
- Limpa (coloca a '0') a *flag* **PIE** do registo de estado **STATUS** para que não surgem mais interrupções durante o atendimento desta. Note-se que, quando o atendimento da interrupção terminar, o conteúdo do registo **ESTATUS** é copiado para o registo **STATUS** e consequentemente as interrupções voltam a estar ativas. Existem arquiteturas que suportam interrupções recursivas no entanto são necessários muitos recursos, nomeadamente *stack*, o que por muitas vezes compromete as performances dos mesmos.
- Guarda o conteúdo do registo **PROGRAM_COUNTER** para o registo **EXCEPTION_RETURN_ADDRESS (EA)** para que depois do atendimento da interrupção o processador possa retornar o processamento da aplicação.
- Depois de salvaguardado o contexto anterior, o processador salta para o endereço da exceção ocorrida colocando este endereço no registo **PROGRAM_COUNTER**.
  Caso se trate de uma interrupção, normalmente tem um salto para a rotina de atendimento *ISR*.

Os endereços de *reset* e do vetor de exceções são definidos na instanciação do processador na ferramenta *SOPC Builder* [6].

Não sendo um controlador vetorizado, a rotina *ISR*, terá de verificar no registo **IPENDING** qual, ou quais, das interrupções está ativa e determinar qual a atender.
Depois de atendida a interrupção o processador deverá voltar para o processamento anteriormente interrompido cujo endereço fora salvaguardado no registo **EXCEPTION_RETURN_ADDRESS (EA)** recorrendo à rotina **exception return** (**ERET)**, especializada para o efeito. Caso surja um novo pedido de interrupção antes do retorno anterior, ou seja antes da execução da rotina **ERET** a rotina de atendimento *ISR* deverá decrementar o valor do registo **EA** de 4.
Ao contrário do *EIC (External Interrupt Controller)*, o *IIC (Internal Interrupt Controller)* está disponível em todas as versões do processador *Nios II*.

2.1.2 - Implementação

Para implementar um sistema que recorra a esta *API* vamos criar um sistema com um porto de entrada, fisicamente ligado a um *Push-button*, que fará *toogle* sobre um *led* da placa *DE-115*. Desta forma, o *Push-button KEY[1]* será uma entrada de interrupção do sistema e o *Push-button KEY[0]* será a entrada de *reset* do sistema conforme nos sugere a figura com a atribuição de *pins* da placa DE-115.

Table 4-2 Pin Assignments for Push-buttons

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
KEY[0]	PIN_M23	Push-button[0]	Depending on JP7
KEY[1]	PIN_M21	Push-button[1]	Depending on JP7
KEY[2]	PIN_N21	Push-button[2]	Depending on JP7
KEY[3]	PIN_R24	Push-button[3]	Depending on JP7

Figura 7 - Push-buttons da placa DE-115 (Fonte: [4])

A instanciação do sistema é feita na ferramenta *SOPC Builder* [6] onde os módulos pretendidos poderão ser adicionados conforme o sistema que se deseje implementar.

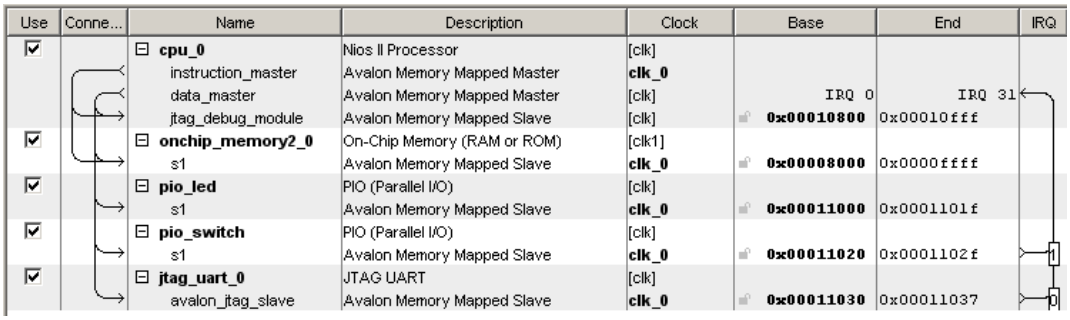

Figura 8 - Sistema com porto PIO para interrupção (Fonte: [6])

Na figura que se antecede podemos então verificar um sistema composto pelo processador *NIOS*, um porto que irá ligar a entrada de interrupção ao *push-button* da placa e outro porto de saída para ativar o *led*.
Note-se que a entrada de *reset* do processador *NIOS* é ativa a *low* o *Push-button* é o mecanismo ideal uma vez que é implementado à custa de um *pull-up*.

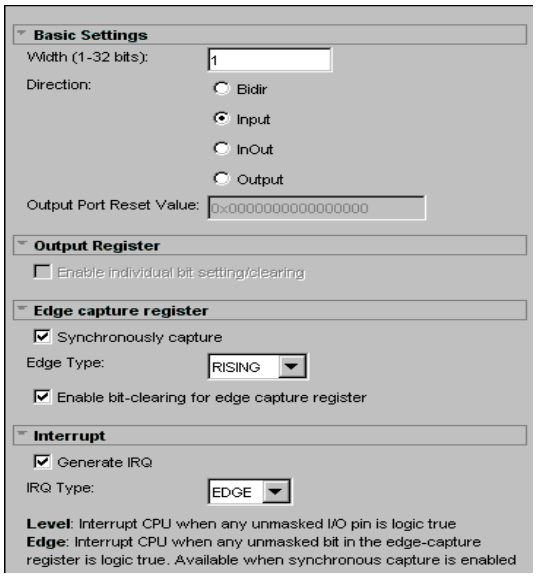

Figura 9 - Configurações do porto PIO (Fonte: [6])

O porto de entrada **pio_switch** foi configurado no modo *edge RISING*, isto significa que este irá ativar a saída de interrupção sempre que se verificar uma transição ascendente

na entrada, o que acontece quando deixamos de pressionar o *Push-button KEY[1]* da placa.
A ativação da opção **enable bit-clearing for edge capture register** tem uma influência determinante na forma como a rotina de atendimento da interrupção irá efetuar *acknowledge*, através da *API*, para que o porto desative o sinal de pedido de interrupção. Com esta *flag* ativa o *acknowledge* é feito escrevendo sobre o registo *edge* do componente *PIO*.
A ferramenta *SOPC Builder* irá gerar a entidade **nios2** sendo, de seguida, necessário criar uma entidade de topo que garanta a ligação dos *pins* da placa aos respetivos portos da entidade anterior.
Desta forma, e em conformidade com o comportamento do controlador de interrupções, a função **main** terá de recorrer à *API* deste, controlador para tornar a sua utilização efetiva.

Por outro lado, tendo em conta que as interrupções irão ser geradas por intermédio de um porto, de entrada, *PIO* teremos de recorrer à sua *API* para ativar as interrupções, e efetuar *acknowledge* na rotina de atendimento.

A implementação da aplicação para atendimento de interrupções consta no Anexo 1 deste relatório.

A utilização da *API* para implementação do atendimento de interrupções pode ser consultada, com mais detalhe em [5] e [8].
No que diz respeito à *API* do porto *PIO*, bem como os registos intervenientes, podemos consultar [14]. Pois, note- se, que a implementação de interrupções recorrendo ao *IIC* implica um conhecimento prévio da sua *API* mas, também, do componente que irá ligar à entrada de interrupção.

2.2 - Módulo Timer

Este componente consiste em dois ou quatro, registo a 16 bits, que pode ser inicializado com um determinado valor e decrementado ao ritmo desejado. Desta forma, conhecendo o período do sinal de *clock* do sistema é possível implementar um *timer* com o período desejado.

O componente *timer* [3] é constituído por 6 registos pelos registos dos quais se destacam os registos **status** e **control**.

O registo **status**, responsável por dar à aplicação o *feedback* em relação ao estado do componente, é constituído pelas *flags* **T0** e **RUN** que indicam se o registo **counter** chegou a zero e se este registo está a contar. A *flag* **T0**, quando ativa, permanece ativa até que a aplicação escreva zero sobre o registo **status**.

O registo **control**, responsável pela configuração do modo de operação do *timer*, é constituído por quatro *flags* conforme nos sugere a figura que se segue.

Registo de controlo do timer	
IT0	Quando no valor lógico ‘1’ ativa a saída *IRQ* do componente.

CONT	Determina o modo como o timer funciona quando o registo **counter** chega a zero. Quando no valor lógico '1' o timer funciona no modo contínuo.
START	Colocando o valor lógico '1' neste *bit*o *timer* começa a contar. Este *bit* é apenas de escrita.
STOP	Colocando o valor lógico '1' neste *bit*o *timer* pára de contar. Este *bit* é apenas de escrita.

Figura 10 - Registo control do módulo Timer

Quando a funcionar no modo contínuo *count-down*, o registo **counter** é decrementado ao ritmo do *clock* do sistema dividido pelo valor do registo **period** e, sempre que este chega a zero, é inicializado com o valor do registo **period** e ativada a saída *IRQ*.

Esta saída do *timer* pode ser utilizada para gerar pulsos retangulares com um período igual à razão entre o *clock* do sistema e o conteúdo do registo **period**. Desta forma, sempre que o registo **couter** chegar a zero a saída **timeout** é ativa durante o período de um ciclo do sinal de *clock* do sistema. O sinal gerado terá então um *duty cycle* de 1/ **period**.

2.2.1 - Relógio de sistema

O relógio de sistema é implementado recorrendo a um *timer* configurado no modo contínuo e a atualização dos parâmetros do sistema relacionados com o relógio são atualizados por intermédio de um *handler* registado no momento de arranque do sistema.

Quando criamos um sistema, no *SOPC Builder*, e incluímos um timer este é detetado pelo *BSP* na criação do projeto e é automaticamente definido como relógio de sistema no *header system.h* através da *macro ALT_SYS_CLK*. O programador deve ter o cuidado de não alterar as configurações deste *timer*, muito menos registar um *handler* para atendimento de interrupções.

O *timer* que implementa o relógio de sistema é inicializado na função **alt_sys_init()** através da *macro ALTERA_AVALON_TIMER_INIT*, definida no *header* ***altera_avalon_timer.h***, que, depois de efetuar alguns testes a verificar se o dispositivo *timer* reune os requisitos necessários, recorre à função **alt_avalon_timer_sc_init()** para configurar o *timer* e registar o *handler* **alt_avalon_timer_sc_irq** que atualiza os parâmetros do relógio de sistema, ambas implementadas no ficheiro ***alt_avalon_timer_sc.c***.

A função **alt_avalon_timer_sc_init()** recorre ao à *API* do *timer* para configurar o registo **control** do timer e à *API* do controlador de interrupções para registar o *handler* **alt_avalon_timer_sc_irq**, que sinaliza o sistema sempre que existir um novo *tick* através da função **alt_tick()**.

O relógio de sistema, quando configurado, dá suporte às funções **gettimeofday()**, **settimeofday()**, e **times()** utilizadas em plataformas UNIX bem como **alt_nticks()**, que retorna o número de *ticks* ocorridos desde o arranque do sistema e **alt_ticks_per_second()** que retorna o número de *ticks* por segundo.

Para testar as funcionalidades do relógio de sistema, bem como a configuração do *timer,* vamos recorrer à funcionalidade do *HAL* **alarm** que permite registar uma função para ser executada a cada número de *ticks* configurados no registo do **alarm**.

Desta forma, e voltando ao exemplo dos *leds*, o *timer* sintetizado será inicializado pelo *HAL*, tal como mencionado anteriormente, em função dos parâmetros inseridos no *SOPC Builder*. A função global **toogle_leds**, anteriormente registada como *handler* da interrupção do *timer*, será registada como **alarm** por forma a efetuar *toogle* sobre o *ledg* ao ritmo de um segundo.

A configuração do componente *timer* instanciado no *SOPC Builder* irá desabilitar as saídas do componente, funcionar no modo contínuo e com uma resolução de 10^{-3} Segundos.

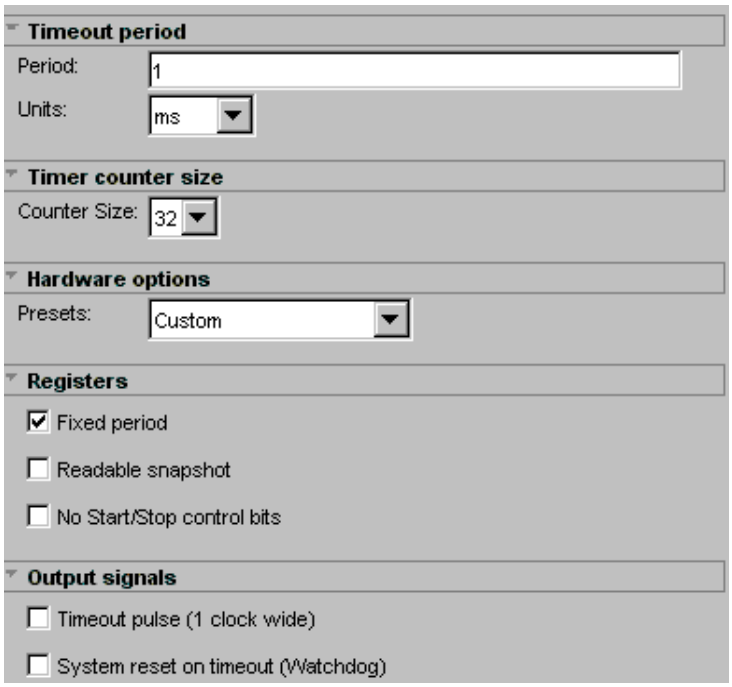

Figura 11 - Configuração do relógio de sistema (Fonte: [6])

Uma vez que a função **alt_avalon_timer_sc_init()** não atualiza o registo **period** do *timer* devemos selecionar a opção **Fixed period** para que este valor seja afetado em tempo de síntese.

A implementação da aplicação para teste do relógio de sistema consta no Anexo 2 deste relatório.

2.3 – Ligação à Memória Externa SDRAM

A placa *DE2-115* está munida com dispositivos *SDRAM* de 64*MB* cada com palavras de 16 *bits*, ou seja, $32M \times 16$ conforme nos sugere a figura que se segue.

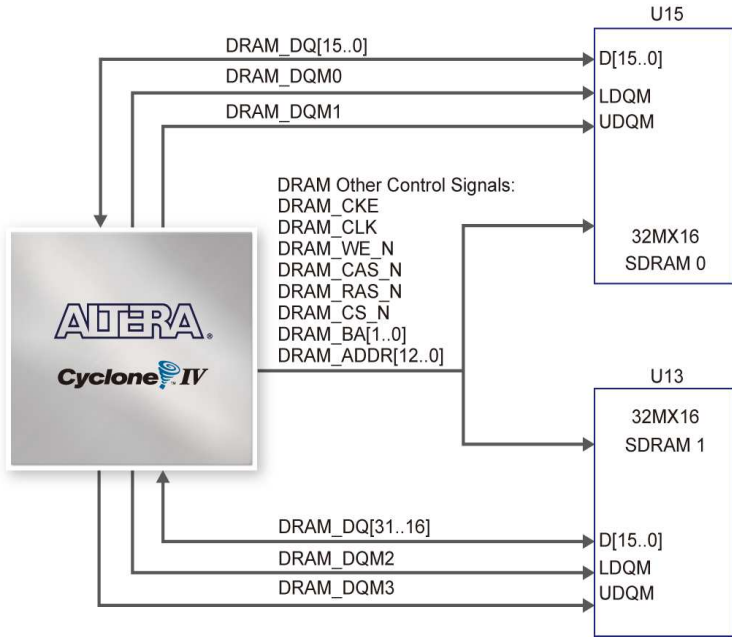

Figura 12 - Dispositivo SDRAM (Fonte: [4])

Ambos os dispositivos *SDRAM* estão ligados à *FPGA* por forma a disponibilizar palavras a 32 *bits* através da concatenação do barramento de dados e partilha do barramento de endereço dando origem à configuração $32\text{M}\times 4=128\text{MBytes}$. O espaço de endereçamento de cada dispositivo *SDRAM* é dividido em 4 devido à presença dos dois *bits* que implementam o conceito de banco. Por sua vez, cada banco, é endereçado num conceito de linha e coluna mapeados através dos *bits* de controlo **DRAM_CAS_N** e **DRAM_RAS_N** de 10 e 13 *bits* respetivamente.

A integração destes dispositivos no sistema irá ser implementada recorrendo à instanciação, no *SOPC Builder*, do componente **sdram_0** que desempenha funções de controlador para este dispositivo.

2.3.1 - Instanciação do módulo SDRAM

A instanciação do módulo **sdram_0**, que desempenha as funcionalidades de controlador para os dispositivos *SDRAM*, é feita através da ferramenta *SOPC Builder*.

O módulo **sdram_0** irá disponibilizar todos os sinais de controlo para os dispositivos ***sdram*** com exceção do sinal de *clock* que é disponibilizado pela placa, tal como tem sido prática nos restantes componentes do sistema.

As figuras que se seguem ilustram o processo de instanciação do módulo.

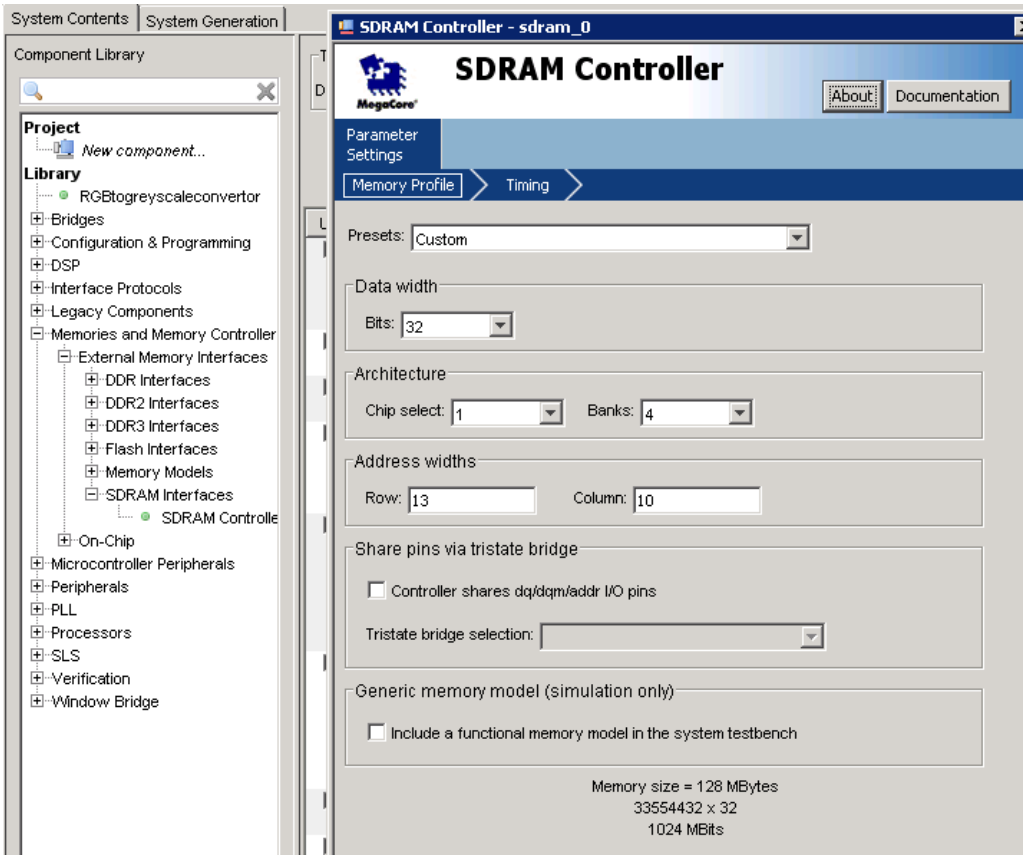

Figura 13 - Componente sdram (Fonte: [6])

Ao adicionarmos o módulo **sdram_0** ao sistema, recorrendo ao navegador em **Controllers > SDRAM Interfaces > SDRAM Controller**, o *SOPC Builder* apresenta um *Wizard* com as características do dispositivo a controlar. Para o sistema em desenvolvimento iremos optar por uma configuração de palavras a 32 *bits*, pelas razões já mencionadas, e um barramento de endereço a 13 *bits* em conformidade com a arquitetura disponibilizada pela placa que permite a descodificação de $2^{13+10} = 8\text{M}$ de endereços por banco e consequentemente, $4 * 8M * 4Bytes = 128MBytes$ no total.

Note-se que devido à decisão tomada em utilizar palavras de 32 *bits* o sistema apenas irá ver um único dispositivo *SDRAM* pelo que apenas necessitamos de um *bit* de controlo para ***Chip Select***.

Após a instanciação do módulo **sdram_0** teremos de reconfigurar o **cpu_0** apontar o endereço de *reset* e vetor de exceções para o espaço de memória utilizado por este módulo.

Depois de instanciado o módulo **sdram_0**, a entidade **nios2**, que descreve o sistema, irá apresentar portos adicionais relativos a este módulo conforme nos sugere o excerto do ficheiro *nios2.vhd*.

A entidade de topo terá de ser alterada de modo a fornecer a conectividade necessária entre os portos do sistema relativos ao componente **sdram_0** e os respetivos *pins* da placa *DE-115*.

A adaptação da aplicação à nova arquitetura consiste em adaptar o *HAL*, que é disponibilizado pelo *BSP*, à nova arquitetura anteriormente alterada. Esta alteração consiste meramente em editar o *BSP* de modo a que a memória *RAM* do sistema passe a estar mapeada para o módulo **sdram_0**.

3.1 - Módulo Ethernet

A placa *DE2-115* está munida com o módulo *Marvell* ***88E1111 que disponibiliza as interfaces MAC GMII/MII/RGMII/TBI*** **e** ***MDIO (Management Data Input/Output)*** que asseguram a transferência de tramas *Ethernet* e respetiva gestão.

O módulo *Ethernet* pode ser implementado recorrendo ao componente **triple_speed_ethernet** (TSE) [20] que implementa as interfaces necessárias para ligar ao periférico e acesso à aplicação através de um *device driver* que pode ser implementado recorrendo ao *HALL* disponibilizado pelo *BSP* para o componente.

3.1.1 - Componente triple_speed_ethernet (TSE)

A instanciação deste componente permitirá, ao programador, uma abstração à camada física do periférico e, com o conhecimento prévio da *API* disponibilizada pelo *device driver*, a aplicação apenas terá de inicializar o dispositivo e configurar os parâmetros *Ethernet* e *IPv4*.

O módulo *TSE* disponibiliza as interfaces *MAC GMII/MII/RGMII/TBI*, para transferência de tramas *Ethernet*, e *MDIO (Management Data Input/Output)*, para gestão do periférico.

O componente descrito pelo *SOPC Builder* terá de disponibilizar a interface *GMII/MII/RGMII/TBI para* que a entidade de topo proceda à ligação destas à placa através dos respetivos pins.

A placa *DE2-115* está configurada, por omissão, para operar no modo *RGMII*, que suporta os três débitos pretendidos pelo que será esta o interface a escolher no *SOPC Builder*.

Tendo em conta de que, do lado do sistema, a interface que este componente disponibiliza para acesso aos *FIFOs* (de transmissão e receção) não permitem um acesso direto à aplicação iremos recorrer a um componente **sgdma** [14], baseado em *DMA*, para dar suporte ao envio e receção das tramas, conforme nos sugere a figura que se segue onde consta o diagrama de blocos do sistema.

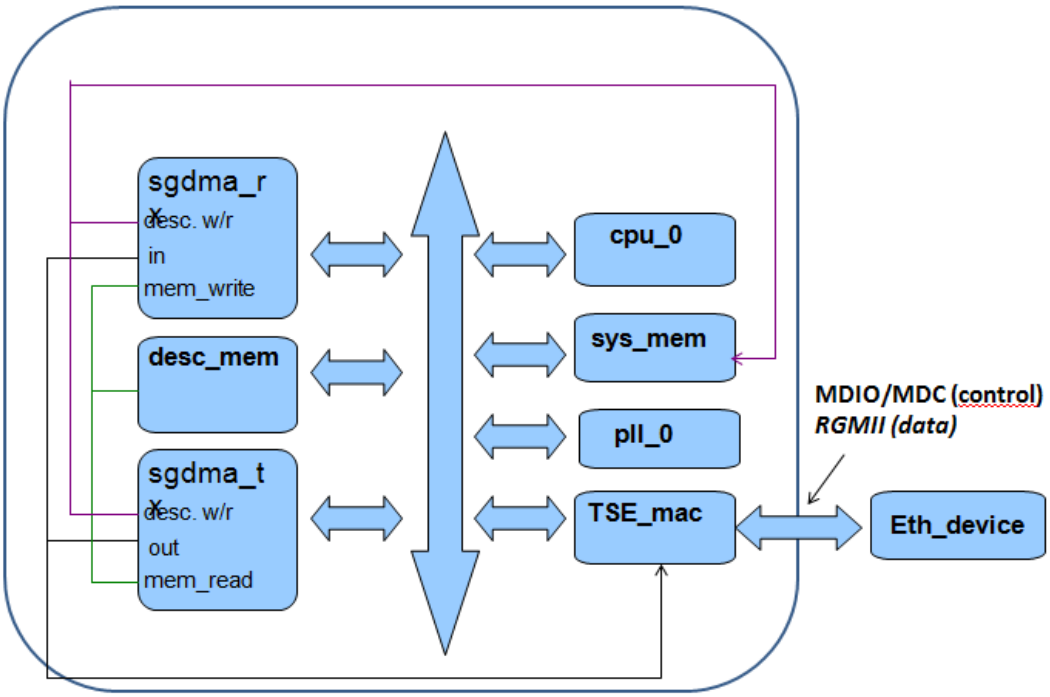

Figura 14 - Diagrama de blocos do sistema com Módulo Ethernet

O componente **sgdma** [14] transfere as tramas *Ethernet* de, e para, a zona de memória (**desc_mem**) cujo endereço consta no descritor em função da operação a efetuar ser de leitura (que consiste no envio de uma trama para o *fifo* do componente **triple_speed_ethernet**) ou de escrita (que consiste na receção de uma trama do *fifo* do componente **triple_speed_ethernet**).

A gestão dos descritores a utilizar no componente **sgdma**, para envio e receção de tramas *Ethernet*, irá ser abordada na implementação do *device driver* do módulo *Ethernet* pelo que aqui interessa perceber que os descritores irão residir na memória do sistema enquanto que a zona de memória para armazenar as tramas *Ethernet*, em ambos os sentidos, será definida no arranque do dispositivo dando a possibilidade, ao programador, de estas residirem em memória dedicada, ou não.

3.1.2 - Entidade de topo

A informação relativa à ligação do periférico *Ethernet* à placa pode ser consultada em [4], na página 58, não dispensando uma consulta mais detalhada ao *datasheet* do periférico.

Informação detalhada acerca do componente **triple_speed_ethernet** pode ser também consultada em [20] onde se descrevem os sinais de controlo passados para a entidade de topo para que se possa efetuar a correspondência aos *pins* do periférico. Pois note-se que este componente não é específico para o periférico aqui em uso e, como tal, devemos consultar este manual. Neste manual merece especial relevo o tópico ***Connecting MAC to External PHYs*** (Página 58) onde são explicados os sinais e a forma como estes devem ser ligados ao periférico *Ethernet*.

A entidade de topo a descrever terá de garantir a correspondência entre os *sinais de controlo* do sistema **nios2**, referentes ao componente **triple_speed_ethernet**, com os *pins* do módulo *Ethernet* não esquecendo a implementação de um *bus MDIO* bidirecional para o lado do módulo *Ethernet* controlado pelo sinal de controlo **mdio_oen** do componente.

Adicionalmente, o sinal de *clock* para a transmissão será fruto da multiplexagem das saídas da *PLL* **pll0** em função dos débitos negociados, que são refletidos nos sinais de controlo **ena_10** e **eth_mode**.

3.1.3 – Device Driver

À semelhança dos periféricos anteriores, o *BSP* disponibiliza uma camada de abstração para os componentes **triple_speed_ethernet** [20] e **sgdma** [14] com as funções base para acesso aos mesmos e que irão servir de base para a implementação deste *device driver*.

A ideia de implementar este *device driver* consiste em adicionar uma camada de abstração para que a camada superior veja este dispositivo como um *Stream Device* e como tal, necessite de apenas duas funções (**read** e **write**) para enviar e receber tramas.

As funções base anteriormente referidas baseiam-se na estrutura **np_tse_mac**, definida no ficheiro ***triple_speed_ethernet_regs.h***, que reflete os registos *MDIO* mapeados em memória do componente **triple_speed_ethernet**. O manual deste componente ***Triple-Speed Ethernet MegaCore Function User Guide***, mais concretamente em [29] na página 71, apresenta uma descrição detalhada de cada um dos registos de onde se realça o registo ***command_config*** que contém as configurações mais relevantes.

A implementação deste *device driver* consta no Anexo 3 deste relatório.

3.2.1 - Variáveis globais

O *device driver* dispõe de duas variáveis globais sendo a primeira ***tse_mac_device*** um *array* de estruturas do tipo ***alt_tse_system_info***, definida no *header altera_avalon_tse.h*, que contém as características do dispositivo *Ethernet*, nomeadamente o endereço base, que consiste no endereço onde estão o registos e o endereço *MDIO* do mesmo.

A segunda variável global corresponde ao objeto do tipo **eth_dev**, instanciada pelo sistema através da macro *MARVELL_88E1111_INSTANCE*, que representa o dispositivo *Ethernet* e corresponde a um encapsulamento do tipo **alt_dev**.

```
…
typedef struct eth_dev_struct
{
        alt_dev dev;
        int iface;
        alt_u8 dev_addr[6];
        tse_mac_trans_info mi;
```

```
        eth_rx_queue rx_queue;
        eth_tx_queue tx_queue;
        int queue_mem_base;
        int queue_mem_span;
        alt_u8 debug_level;
        alt_u8 cfg;
        //sinalize_tcp_ip_stack sinalize_stack;
        eth_mib mib;
}eth_dev;
…
#define MARVELL_88E1111_INSTANCE(name, dev) \
static eth_dev dev =                        \
{                                           \
  {                                         \
    ALT_LLIST_ENTRY,                        \
    "/dev/tse_0",                           \
    NULL, /* open */                        \
    eth_close, /* close */                  \
    eth_rcv, /* read */                     \
    eth_send, /* write */                   \
    NULL, /* lseek */                       \
    NULL, /* fstat */                       \
    NULL, /* ioctl */                       \
  },                                        \
…
}
```

Lista de Código 1 - Variáveis globais - source marvell_88e111.h

O tipo **alt_dev**, definido pelo *HAL* dispõe, nos seus atributos, de apontadores para as funções *IO* que são inicializadas com as funções implementadas no *device driver* dando origem a uma *API standard* de qualquer *stream device*.

O *header* do *device driver* também define as estruturas necessárias para o controlo do dispositivo que são encapsuladas pela estrutura ***eth_dev*** instanciada como variável global no ficheiro fonte (implementação) *marvell_88e111.c*. Neste podemos verificar o endereço *MAC* a atribuir ao dispositivo *Ethernet*, os *QUEUES* de receção e transmissão e o driver anteriormente referido.

Nesta versão do *device driver* o *QUEUE* de transmissão (*eth_tx_queue*) é composto apenas por um descritor com o intuito de privilegiar as tramas recebidas uma vez que a aplicação poderá controlar o fluxo neste sentido. No que diz respeito ao *QUEUE* de receção (*ethrtx_queue*), este é constituído por uma pilha de lista de descritores que vão sendo atualizados pelo *device driver* e consumidos pela aplicação com recurso à estrutura ***eth_packet_mux***.

3.2.2 - Função eth_open_dev

A função **eth_open_dev** tem como objetivo fornecer à aplicação uma forma de aceder ao dispositivo sem qualquer conhecimento do mesmo para além do seu nome, neste caso concreto "/dev/tse_0".

3.2.3 - Função eth_init

A função ***eth_init*** tem como objetivo inicializar os componentes inerentes ao módulo

Ethernet e deve ser evocada apenas uma vez.

Esta recebe, como argumento, um apontador para o dispositivo retornado pela função anterior e informação referente à localização de memória onde as tramas *Ethernet* irão ser armazenadas.

Note-se que esta localização de memória está seriamente comprometida com a arquitetura do sistema.

A associação do módulo *Ethernet* ao *device driver* é feita através da indexação no *array* implementado pela variável global ***tse_mac_device*** onde a função ***alt_tse_system_add_sys***, definida no *header altera_avalon_tse.h*, inicializa os parâmetros do módulo.

Conforme referido anteriormente, o sistema está configurado para funcionar no modo *RGMII*. A comutação para este modo é feita recorrendo à função ***marvell_cfg_rgmii***, definida no *header altera_avalon_tse.h*, que manipula os registos 14 e 1b que estão mapeados em memória pela estrutura ***np_tse_mac***.

Esta função também é responsável por abrir e inicializar os componentes **sgdma** que, tal como mencionado anteriormente, servem de interface entre os *FIFOS*, de transmissão e receção do componente **tse_0**, e o sistema. O *BSP* (*Board Support Package*) disponibiliza uma *API* com o *HAL* deste componente através do *header altera_avalon_sgdma.h* sendo a abertura do componente feita através da função ***alt_avalon_sgdma_open***.

A função ***tse_mac_initTransInfo2***, definida no *header altera_avalon_tse.h*, apenas inicializa a estrutura ***tse_mac_trans_info*** que contém os apontadores para os componentes **sgdma**.

A inicialização efetiva dos componentes **sgdma** é feita na função ***eth_sgdma_init*** que será alvo de estudo no tópico que se segue.

3.2.4 - Função eth_sgdma_init

Tal como já foi referido, na abordagem à arquitetura do sistema, os componentes **sgdma** irão assegurar a interface entre os *FIFOS*, de transmissão e receção das tramas *Ethernet*. O documento *Embedded Peripherals IP User Guide.pdf* [14] pode ser consultado para obter informação mais detalhada acerca da *API* do *HAL* que o *BSP* disponibiliza para este componente.

A função ***eth_sgdma_init*** implementa a inicialização efectiva dos componentes **sgdma** e dos respetivos descritores que irão ser consumidos pela transferência de tramas *Ethernet*. Esta recorre aos *headers altera_avalon_sgdma_regs.h*, que descreve os registos mapeados em memória, e *altera_avalon_sgdma.h*, que declara as funções da *API*.

A inicialização do componente **sgdma** é feita manipulando os registos *CONTROL* e *STATUS* deste componente.

A transmissão de tramas *Ethernet* será feita de forma síncrona enquanto a receção será

feita de forma assíncrona recorrendo a um mecanismo de atendimento de interrupção para processamento das mesmas.

A inicialização dos descritores consiste em alocar memória necessária para armazenar uma trama *Ethernet* recorrendo à função ***eth_desc_mem***, que retorna o respetivo apontador, inicializar o descritor recorrendo à função ***alt_avalon_sgdma_construct_stream_to_mem_desc*** e configurar o registo *CONTROL*.

No caso da receção, a inicialização dos descritores, é feita de forma a ligar cada um destes ao seguinte, formando uma lista aberta.

3.2.5 - Função eth_config

A função ***eth_config*** tem como objetivo configurar os componentes inerentes ao módulo *Ethernet* e deve ser evocada sempre que se pretenda alterar algum destes parâmetros, nomeadamente o endereço *mac* da interface *Ethernet* ou mesmo para, por exemplo, colocar a interface a funcionar em modo promíscuo. Sempre que a aplicação evocar a função ***eth_config***, esta, irá configurar os registos *MAC* do módulo *Ethernet* e evocar a função ***eth_open***, que não está disponível para acesso direto da aplicação, para configurar os componentes **sgdma**.

A implementação desta função recorre à implementação do *HAL* para o componente **tse_0** disponibilizado através do *header altera_avalon_tse.h* e do *header triple_speed_ethernet_regs.h*, que define as *macros* de acesso aos registos do componente.

A função ***getPHYSpeed*** implementa o processo de auto negociação e retorna informação relevante acerca do débito 10/100/1000 e do modo *duplex* bem como eventuais causas para a ocorrência de erros decorridos durante o processo.

A variável local ***cmd_config*** é afetada com o valor atual do registo ***command_config*** através da sua leitura recorrendo à *macro* ***IORD_ALTERA_TSEMAC_CMD_CONFIG*** para posterior alteração função do valor retornado pela função ***getPHYSpeed*** e atualização do registo através da *macro* ***IOWR_ALTERA_TSEMAC_CMD_CONFIG***.

O endereço *mac* da interface é configurado através das *macros* ***IOWR_ALTERA_TSEMAC_MAC_0*** e ***IOWR_ALTERA_TSEMAC_MAC_1*** que afetam os respetivos registos mapeados em memória.

3.2.6 - Função eth_sgdma_open

A função ***eth_sgdma_open*** não está disponível à aplicação e é evocada pela função ***eth_config*** para configurar o processamento de descritores, por parte dos componentes **sgdma**, de transmissão e receção, sempre que a interface é configurada.

No sentido de receção o componente **sgdma** é configurado por forma a gerar pedidos de interrupção sempre que, entre outros, um descritor for consumido através da *macro* ***IOWR_ALTERA_AVALON_SGDMA_CONTROL*** que atualiza o registo *CONTROL* do componente. Adicionalmente é efetuado o registo da rotina *ISR (Interrupt Service Routine)*, que irá processar os pedidos de interrupções, através da função ***alt_avalon_sgdma_register_callback***.

A transferência assíncrona, que corresponde à cópia das tramas *Ethernet* dos *FIFO* de receção para os descritores através do componente **sgdma**, é iniciado através da função ***tse_mac_aRxRead***.

Adicionalmente, a função ***eth_open***, configura o registo **command_config** em função dos parâmetros configuráveis através do atributo ***cfg*** disponibilizado à aplicação e dá início ao processo de transmissão e receção de tramas por parte do módulo *Ethernet.*

3.2.7 - Função eth_close

A função ***eth_close*** é responsável por desabilitar a receção de tramas *Ethernet* por parte do módulo. Tal é conseguido desinstalando o *handler* de interrupção no módulo *SGDMA* de receção e efetuando *clear* sobre a *flag* ***RX_ENA*** do registo ***command_config*** para desabilitar a receção de tramas por parte do módulo *Ethernet*.

3.2.8 - Função eth_rx_sgdma_isr

A função ***eth_rx_sgdma_isr*** é evocada sempre que ocorrer uma transferência de tramas *Ethernet* entre o *FIFO* de receção do componente **tse_0** e os descritores através do componente **sgdma_rx**, e quando uma lista de descritores for consumida por completo. O componente **sgdma_rx** verifica o fim da lista testando a *flag* ***OWNED_BY_HW*** do registo ***CONTROL*** do último descritor da lista.

Quando é gerada uma interrupção, fruto da transferência de uma trama *Ethernet* entre o *FIFO* de receção do componente **tse_0** para um descritor do componente **sgdma_rx**, são feitos dois testes nomeadamente ***desc_comp***, que determina se houve receção de trama, e ***chain_comp***, que determina se a *chain* de descritores terminou. Estes testes são efetuados sobre o registo *STATUS do* componente **sgdma_rx**, mais concretamente às *flags* ***DESC_COMPLETED, EOP_ENCOUNTERED*** e ***CHAIN_COMPLETED***.

No caso de receção de trama *Ethernet* é feita uma verificação de ocorrência de erros através do registo *STATUS* do descritor pelo que caso se confirme é incrementado o contador de erros e a trama é descartada colocando a *flag* ***OWNED_BY_HW*** no valor lógico "1" para que a função ***eth_rcv*** não retorne a trama para a aplicação, pois esta foi a forma adotada para implementar o controlo das tramas descartadas. Após a verificação de ocorrência de erros o contador ***chain_loop***, que serve para iteração na *chain*, é incrementado para que haja um controlo, por parte do *driver*, sobre os descritores que já foram processados.

Quando a *chain* de descritores termina o contador ***list_loop*** é incrementado para que se dê início ao processamento de uma nova *chain* dentro da ***chain_list***. Sempre que se verifica uma mudança de *chain* de descritores, esta é percorrida com o intuito de verificar se a trama em causa foi retornada para a aplicação através da *flag* ***OWNED_BY_HW*** uma vez que a função ***eth_rcv*** coloca esta *flag* valor lógico "1" nas tramas que retorna para a aplicação.

3.2.9 - Função eth_send

A função ***eth_send*** implementa as transferências de tramas da aplicação para a rede *Ethernet.* Não é bloqueante e apenas processa a transferência caso tenha condições para tal pois, caso contrário, como por exemplo o componente **sgdma_tx** esteja com

uma transferência em curso, a função retorna imediatamente. Nestes casos compete à aplicação efetuar o controlo de fluxo.

Uma segunda *release* para este *device driver* poderia incluir melhorias de performance através da implementação de um *queue*, em todo semelhante ao implementado no sentido da receção, possibilitando o modo assíncrono neste sentido da comunicação. Uma segunda alternativa seria recorrer a uma segunda camada que implementasse um *queue* em ambos os sentidos de comunicação e que implementasse mecanismos de *QoS*.

Por cada transferência verifica-se a inicialização do descritor de transmissão, através da função ***alt_avalon_sgdma_construct_mem_to_stream_desc*** que está definida no *header altera_avalon_sgdma.h*, e a cópia do conteúdo da trama para a zona de memória definida no descritor.

A função ***tse_mac_sTxWrite***, definida no *header altera_avalon_tse.h*, implementa a transferência, no modo síncrono, dos dados do descritor para o *FIFO* do componente **tse_0**.

3.2.10 - Função eth_rcv

A função ***eth_rcv*** implementa as transferências de tramas da lista de descritores para a aplicação. Não é bloqueante e apenas processa a transferência caso a lista de descritores não esteja vazia.

A função ***eth_rx_sgdma_isr***, tal como pudemos verificar, recorre aos contadores ***list_loop*** e ***chain_loop*** para manter controlo sobre qual o descritor que está a ser consumido pelo componente **sgdma_rx**. A função ***eth_rcv*** irá tirar partido destes contadores e da estrutura ***eth_packet_mux*** para determinar se existem tramas para retornar à aplicação conforme nos sugere o diagrama que se segue.

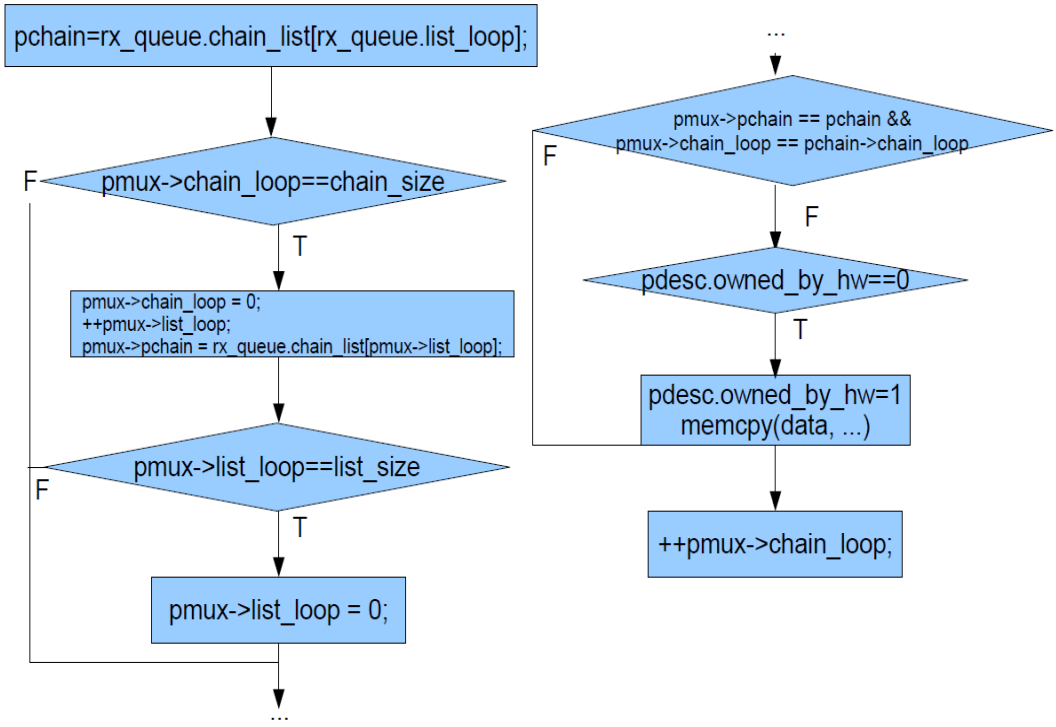

Figura 15 - Pseudo código da função eth_rcv

Analisando o diagrama da função ***eth_rcv***, esta verifica se o iterador da *chain* de descritores da estrutura ***eth_packet_mux*** atingiu o seu valor máximo para verificar a necessidade de passar para a lista seguinte. Caso se confirme, a passagem para a lista seguinte consiste em incrementar o iterador da lista ***list_loop*** e inicializar o iterador da *chain* ***chain_loop***.

O teste seguinte consiste em verificar a possibilidade de a estrutura ***eth_packet_mux*** estar a apontar para a *chain* de descritores que está atualmente a ser consumida pelo componente **sgdma_rx**. Nesta implementação do *driver* as tramas da *chain* que está a ser consumida pelo componente apenas são retornadas para a aplicação quando esta estiver completa.

Desde que a *chain* de descritores atualmente apontada pela estrutura ***eth_packet_mux*** esteja descomprometida com o componente, as tramas contidas serão retornadas para a aplicação.

3.1.4 - Aplicação arp request (a numeração deste header não deve ser 3.3…)

A aplicação ***arp-request*** foi implementada com o intuito de testar o *device driver marvell_88e111* e consiste no envio, sistemático, de um pedido de *arp request* para a rede por forma a determinar o endereço *mac* de um *host* com determinado endereço Ipv4.

A implementação desta aplicação consta no Anexo 4 deste relatório.

A função ***create_arp_request*** implementa a formatação da trama *Ethernet* e pacote *ARP_REQUEST*, em conformidade com o *RFC 826* [23].

A função ***send_arp*** recorre ao *driver* para enviar o pacote *ARP_REQUEST* através do *driver* do módulo *Eternet*.

Para o envio sistemático do pacote *ARP_REQUEST* foi instanciado um objeto ***alarm*** que foi inicializado com a função ***send_arp*** para que o pacote seja enviado para a rede a cada 10 segundos.

A função ***main*** recorre à função ***eth_init*** para inicializar o módulo *Ethernet* e receber o apontador para a estrutura ***eth_driver*** que contém os apontadores para as restantes funções necessárias ao funcionamento do módulo *Ethernet*.

3.2 - Stack TCP/IP – uIP (este deve ser 3.3)

Um *stack TCP/IP* implementa as camadas 2 e 3 do modelo *OSI* que, no contexto de uma rede *Ethernet*, correspondem aos protocolos *IP*, camada de rede, e *TCP*, camada de transporte.

A implementação adotada baseia-se no pacote *open source* **uIP** desenvolvido por Adam Dunkels [18] onde consta, entre outros, informação detalhada acerca deste pacote de *software*.

O **uIP**, apesar de apresentar algumas limitações, é amplamente utilizado em sistemas embebidos dada a sua simplicidade de *porting*, uma vez que o único compromisso que dispõe com a arquitetura é o *device driver* do módulo *Ethernet* e o *timer*.

Na figura que se segue podemos verificar, de uma forma muito superficial, o fluxo de uma possível implementação recorrendo ao núcleo *TCP/IP* do **uIP**.

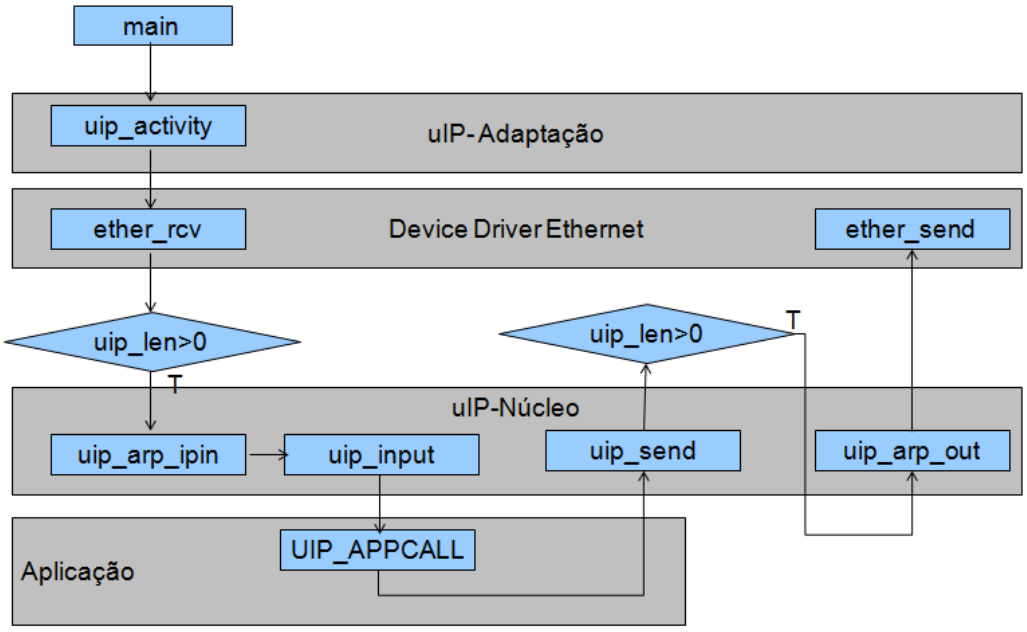

Figura 16 - Implementação do Stack TCP/IP recorrendo ao uIP

As funções que constam no núcleo do **uIP** já estão implementadas e, como tal, serão

apenas evocadas. Desta forma, a implementação do *stack* irá envolver a implementação das funções referentes à adaptação e aplicação.

De uma forma genérica, a função ***main*** irá efetuar *polling* sobre a função ***uip_activity*** que, por sua vez, tem a responsabilidade de interagir com o *device driver* do módulo *Ethernet* para verificar se existem tramas a processar. Caso existem tramas a processar, a função ***uip_activity***, retorna a trama para o núcleo **uIP**, através da função ***uip_input***, e atualiza a *cache* de *arp* através da função ***uip_arp_ipin***.

O núcleo **uIP** está programado para, após processar o cabeçalho do pacote *IP* da trama recebida, evocar a função definida pela *macro* ***UIP_APPCALL*** para passar o pacote para o nível da aplicação. Por sua vez, a aplicação, caso pretenda enviar dados para o *peer* remoto, recorre à função do núcleo **uIP** ***uip_send***.

Depois do controle do processador voltar para a função ***uip_activity***, a variável global ***uip_len***, também definida no núcleo **uIP**, servirá para verificar se a aplicação colocou dados no *buffer* pelo que será evocada a função ***uip_arp_out*** para processar o *header* do pacote *IP* bem como os endereços *mac* da trama e finalmente evocada a função do *device driver* do módulo *Ethernet* para o envio da trama para a rede.

A distribuição do pacote de *software* do **uIP** está estruturada de forma a minimizar o número de ficheiros a alterar para a adaptação à arquitetura e à plataforma. Desta forma todo o compromisso da arquitetura é implementada no ficheiro fonte *uip_nios2.c* e com a plataforma no *header uip-conf.h* conforme nos sugere os dois tópicos que se seguem.

3.2.1 - Adaptação à Arquitetura

O núcleo do **uIP** pressupõe que a arquitetura disponha de recursos para dar suporte de *timer* para que este possa controlar as ligações inativas e manter a consistência na manutenção do estado das mesmas. O segundo pressuposto consiste na existência de um dispositivo físico para comunicar com a rede *Ethernet*.

No que diz respeito ao *timer*, a adaptação consiste em definir o tipo ***clock_time_t*** no *header clock-arch.h*, conforme nos sugere o ficheiro onde consta o conteúdo deste ficheiro.

```
#include <alt_types.h>
typedef alt_u32 clock_time_t;
#define CLOCK_CONF_SECOND 100
```

Lista de Código 2 - uIP Adaptação à arquitectura – clock-arch.h

Toda a restante adaptação, à arquitetura, é feita no ficheiro fonte *uip_nios2.c* onde é definida a função ***clock_time*** que é declarada pelo núcleo **uIP** no *header clock.h*.

```
clock_time_t clock_time(void)
{
        return alt_nticks();
}
```

Lista de Código 3 - uIP Adaptação à arquitectura – uip_nios2.c

Para além dos recursos físicos anteriormente mencionados, o núcleo **uIP**, também pressupõe a existência de duas funções adicionais cuja implementação também deve constar no ficheiro fonte *uip_nios2.c*, nomeadamente a função ***uip_setup***, responsável por configurar o núcleo com os parâmetros *IP* do *host* e pela inicialização do dispositivo de rede, e a função ***uip_activity***, responsável por interagir com o dispositivo de rede por intermédio do seu *device driver*.

Note-se que, uma vez que ambas as funções anteriormente referidas são evocadas fora do núcleo, não existe compromisso com o nome ou parâmetros sendo apenas necessário que estas cumprem os seus pressupostos.

```
int uip_setup(uip_ip_config * ip)
{
	uip_logf(" - TCP/IP Stack uIP initialization Process...\n");
	//timers initialization
	timer_set(&periodic_timer, PERIODIC_TIMER_TIMEOUT);
	timer_set(&arp_timer, ARP_TIMER_TIMEOUT);
	/*
	 * initialize ethernet device
	 */
	eth = (eth_dev*)eth_open_dev(ETHERNET_DEVICE_NAME, ETHERNET_IFACE);
	if(eth == NULL)	return -1;

	if(eth_init(eth, ETHERNET_FRAME_BUFFER_ADDRESS, ETHERNET_FRAME_BUFFER_SIZE, \
	ETH_DEBUG_CONFIG/*|ETH_DEBUG_SGDMA|ETH_DEBUG_ISR*/)!= ETHERNET_SUCCESS)

	return -1;

	if(eth_config(eth, ip->uip_mac_addr.addr) != ETHERNET_SUCCESS)	return -1;
	/*
	 * uip initialization
	 */
	uip_init();
	uip_setethaddr(ip->uip_mac_addr);
	uip_sethostaddr(ip->uip_ip_addr);
	uip_setdraddr(ip->uip_ip_gw);
	uip_setnetmask(ip->uip_ip_mask);
	return 0;
}
```

Lista de Código 4 - uIP Adaptação à arquitetura uip_setup – uip_nios2.c

A função ***uip_setup*** recebe, como argumentos, uma estrutura com os parâmetros *IP* para configurar o *host* recorrendo a funções do núcleo **uIP**, nomeadamente ***uip_setethaddr, uip_sethostaddr, uip_setdraddr*** e ***uip_setnetmask***.

```
static void uip_handle_packets(void)
{
	do
	{
		uip_len = eth->dev.read((alt_fd*)eth, (char*)&uip_buf[0], (int)UIP_BUFSIZE);
		if(uip_len > 0)
		{
			if (ETHER->type == htons(UIP_ETHTYPE_IP))
			{
```

```
                                        //actualize arp cache
                                        uip_arp_ipin();
                                        //Process IP packet - calls uip_process()
                                        uip_input();
                                        //If the above function invocation resulted in data that
                                        //should be sent out on the network, the global variable
                                        //uip_len is set to a value > 0.
                                        if (uip_len > 0)
                                        {
                                                //Process Mac header according arp table
                                                uip_arp_out();
                                                while(eth->dev.write((alt_fd*)eth,(char*)uip_buf,
(int)uip_len) != uip_len){}
                                        }
                                }
                                else if (ETHER->type == htons(UIP_ETHTYPE_ARP))
                                {
                                        //Proccess arp request
                                        uip_arp_arpin();
                                        /* If the above function invocation resulted in data that
                                        should be sent out on the network, the global variable
                                        uip_len is set to a value > 0. */
                                        if (uip_len > 0)
                                        {
                                                eth->dev.write((alt_fd*)eth,(char*)&uip_buf[0],
(int)uip_len);
                                        }
                                }
                        }
                }while(uip_len > 0);
}
```

Lista de Código 5 - uIP Adaptação à arquitetura uip_handle_packets – uip_nios2.c

A função ***uip_activity***, recorre à função ***uip_handle_packets*** para receber as tramas *Ethernet* da rede através do *device driver* do componente **tse_0** e passá-las para o núcleo **uIP** para que estas possam ser processadas e os respetivos segmentos sejam passadas à aplicação.

Caso a aplicação pretenda enviar segmentos terá de recorrer à função do núcleo **uip_send** resultando na afetação da sua dimensão (em *bytes*) na variável global **uip_len**. A função ***uip_handle_packets***, após o retorno do núcleo, deve verificar a existência de tramas para envio e recorrer ao *device driver* para enviar as tramas para a rede *Ethernet*.

3.2.2 - Adaptação à plataforma

A adaptação à plataforma é feita no *header uip-conf.h* e consiste em especificar alguns parâmetros inerentes ao funcionamento do núcleo **uIP**, declarar alguns dos tipos utilizados pelo núcleo e recorrer à *macro UIP_APPCALL* para determinar qual a função que irá ser evocada pelo núcleo para passar os datagramas recebidos para a aplicação.

Note-se que, relativamente aos parâmetros inerentes ao funcionamento do núcleo **uIP**, caso estes não sejam definidos, serão utilizados os parâmetros definidos no *header uipopt.h*, que funcionam como omissão.

```
#ifndef _UIP_CONF_H_
#define _UIP_CONF_H_

#include <inttypes.h>
#include "types.h"

extern void uip_log(char *msg);
extern void uip_logf(const char *msg, ...);

typedef void * uip_tcp_appstate_t;

typedef unsigned short uip_stats_t;
#define UIP_CONF_ARPTAB_SIZE   10
#define UIP_CONF_MAX_CONNECTIONS 4
#define UIP_CONF_MAX_LISTENPORTS 2
#define UIP_CONF_BUFFER_SIZE 420
#define UIP_CONF_BYTE_ORDER LITTLE_ENDIAN
#define UIP_CONF_LOGGING 0
#define UIP_CONF_UDP 0
#define UIP_CONF_UDP_CHECKSUMS 1
#define UIP_CONF_STATISTICS 1
extern void net_handle_connection(void);
#define UIP_APPCALL net_handle_connection

#endif//_UIP_CONF_H_
```

Lista de Código 6 - uIP Adaptação à plataforma – uip-conf.c

3.3 - Interface Socket

Antes de começarmos a abordagem à implementação da interface *Socket* interessa percebermos qual a sua finalidade.

Se tentássemos implementar uma aplicação servidora, como por exemplo um servidor *web*, recorrendo à implementação anterior, ou seja, o **uIP** adaptado à arquitetura aqui em estudo e à plataforma disponível teríamos de implementar uma função ***net_handle_connection*** especializada para o efeito.

Uma vez que uma ligação *http* muito dificilmente é implementável num conceito de comando/resposta teríamos de implementar uma forma de manter a coerência em cada uma das ligações estabelecidas pelos *peers* remotos para que os segmentos fossem enviados para os seus destinos corretos.

O núcleo do **uIP** tem um *array* de estruturas do tipo ***uip_conn*** onde são mantidos os estados das ligações mas a aplicação teria de estar a consultar esta estrutura por cada pacote recebido.

A interface *Socket* tem como finalidade disponibilizar uma *API* agradável de utilizar possibilitando uma abstração total aos desafios anteriormente colocados. Ou seja, serão disponibilizadas à aplicação um conjunto de funções que permitirão uma implementação de aplicações clientes e servidores onde por cada ligação estabelecida existirá um *socket* que identifica a ligação de forma única.

Para a manutenção do estado das ligações iremos recorrer a uma estrutura ***socket*** que

dispõe de todos os atributos necessários para este efeito, conforme nos sugere o excerto que se segue.

```
typedef volatile struct socket_window_struct
{
        u8_t ack_seq[4]; //segment sequence number
        u16_t len;              //Length of the data that was sent
}socket_window;

typedef uip_ip_config net_ip_config;

typedef volatile struct socket_struct
{
  u8_t input_buffer[SOCKET_BUFFER_SIZE];        //used to store incoming data from network
  u8_t output_buffer[SOCKET_BUFFER_SIZE];       //used to store outgoing data to network
  alt_u8 state;                                 //stores current state of the process
  Fbuffer input_fbuffer;                        //input_buffer manipulation
  Fbuffer output_fbuffer;                       //output_buffer manipulation
  struct uip_conn * conn;                       //Related connection in UIP
  /*
   * information related to tcp retransmitions
   * Length of the data that was previously sent is on uip_conn
   */
  u8_t* sent_ptr;           //next segment pointer
  u16_t sent_size;          //next segment size
  socket_window window[SOCKET_WINDOW_SIZE];
}socket;
```

Lista de Código 7 - Estrutura socket

Como podemos verificar no excerto que se antecede, a estrutura *socket* dispõe de dois b*uffer* para guardar o conteúdo dos segmentos recebidos da rede e da aplicação, para posterior envio para a rede. Uma estrutura, do tipo ***fbuffer*** para manipulação dos *buffers* anteriormente referidos e dois atributos, ***sent_ptr*** e ***sent_size*** para dar suporte a pedidos de retransmissão por parte do *peer* remoto. Adicionalmente dispõe de um apontador para a estrutura ***uip_conn*** do *array* do núcleo **uIP** que contém informação da ligação (*struct uip_conn * conn*) e de um *byte* para controlo do estado da ligação (*alt_u8 state*).

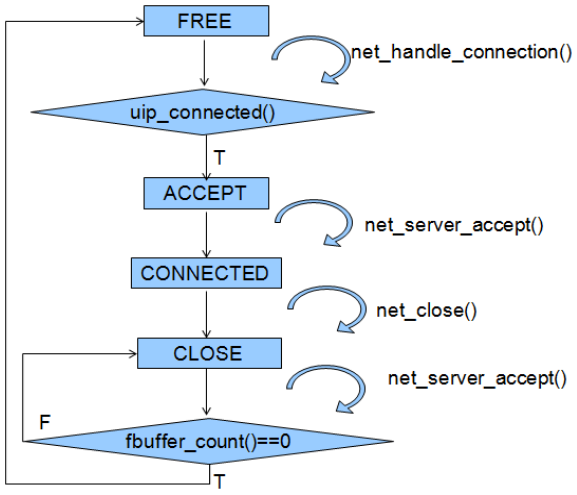

Figura 17 - Estados de uma ligação (Socket)

A figura que se antecede apresenta os estados que uma ligação pode ter bem como as respetivas funções da *API socket* que fazem a transição entre os estados.

Sempre que um pacote chega ao núcleo **uIP**, após processamento do mesmo, estes são passados para a função ***net_handle_connection*** conforme definido pela macro UIP_APPCALL tornando-se evidente a escolha desta função para controlar a máquina de estados do *socket*.

Quando uma nova ligação é estabelecida a função ***net_handle_connection*** é evocada pelo núcleo **uIP** com a *flag* ***uip_connected*** ativa. Neste caso, a função ***net_handle_connection***, percorre o *array* de *sockets* para pesquisar por um *socket* que se encontre no estado *FREE* para passar para o estado *ACCEPT* por forma a sinalizar que este *socket* dispõe de uma ligação à espera de ser atendida pela aplicação servidora.

Compete à aplicação servidora estar ativamente a verificar se existem ligações pendentes para processamento através da função ***net_server_accept*** que pesquisa no *array* de *sockets* por um *socket* no estado *ACCEPT* para retornar e passar o estado deste para *CONNECTED*.

Uma vez no estado *CONNECTED* o *socket* permanecerá neste estado até que a aplicação decida desligar a ligação através da função ***net_close***. A aplicação pode decidir desligar a ligação perante uma situação normal como por exemplo os dados já foram processados e enviados para o *buffer*, mas tendo em conta que a qualquer momento o *peer* remoto pode decidir desligar a ligação, mesmo que a aplicação ainda não tenha terminado todo o processamento, neste caso, a aplicação será sinalizada através do valor retornado pela função ***net_send*** *INVALID_SOCKET*. Esta situação é sinalizada pelo núcleo **uIP** através das *flags uip_closed()* e *uip_aborted()* e, consequentemente, a máquina de estados liberta o respetivo *socket*.

Note-se que a aplicação também deverá ter algum *feedback* acerca do número de *bytes* escritos no *buffer* sempre que efetuar uma operação de escrita neste recurso uma vez que terá de saber lidar com situações de *buffer overflow* (retorno de zero).

A implementação da interface *socket* consta no ficheiro fonte net.c no Anexo 5 deste Relatório.

Capítulo 4 - Módulo D5M

O módulo *D5M* tem como finalidade controlar a Câmara *D5M*, que está ligada placa *DE2-115* através do porto *GPIO*, e processar os pixéis por forma a gerar tramas *RGB*.

A câmara utiliza o porto *GPIO* para transferir os pixéis e os sinais de controlo (*LVAL* e *FVAL*), bem como um *bus I2C* utilizado para configurar os registos internos de configuração da mesma.

Para implementar o controlo e aquisição de imagem da Câmara o módulo irá ser dividido em dois módulos distintos, sendo o primeiro o para efeitos de controlo deste dispositivo e um segundo módulo para a aquisição de imagem, conforme nos sugere a figura que se segue.

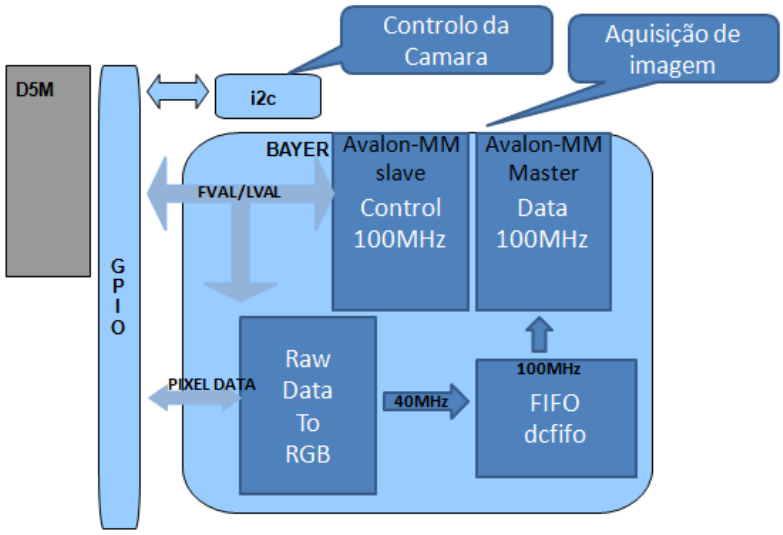

Figura 18 - Estrutura do módulo D5M

4.1 - Módulo de aquisição de imagem

A motivação para a implementação deste módulo deve-se ao facto de a câmara *D5M* debitar os pixéis no formato *bayer* e, como tal, verifica- se a necessidade de implementar uma forma de converter este formato em *RGB* e enviá-lo para o sistema **nios2**.

O módulo aqui implementado terá de ser configurável, para uma mais fácil adaptação ao sistema para onde vai enviar as tramas *RGB*, funcionar em domínios de *clock* distintos, uma vez que a frequência de *clock* do sistema (100MHz) não coincide com a frequência de aquisição dos pixéis.

4.1.1 – Sub Módulo de controlo

Este módulo será responsável pela implementação da máquina de estados bem como pela gestão da interface *Avalon-MM Slave* para gestão dos registos de configuração.

Para o efeito, foram considerados os registos que constam no quadro que se segue.

Registo	**Endereço**	**Acesso**	**Observação**
control_reg	0x1	R/W	Registo de controlo composto por diversas *flags*.
frame_width_reg	0x2	R	Retorna a coluna actual.
frame_height_reg	0x3	R	Retorna a linha actual.
max_frame_width_reg	0x4	R	Número máximo de colunas.
max_frame_height_reg	0x5	R	Número máximo de linhas.
color_depth_reg	0x6	R	Número de *bits* por pixel.
dma_address_reg	0x7	R/W	Endereço de memória do sistema que determina o início do *frame buffer*.
captured_frames_reg	0x8	R	Número de tramas capturadas desde o último *reset*.
max_address_span_reg	0x9	R/W	Dimensão máxima do *frame buffer*.

Figura 19 - Mapa de registos do módulo D5M

O registo de controlo **control_reg** contém as *flags* necessárias para o controlo de todo o módulo. Apesar de o seu controlo ser da responsabilidade do módulo de controlo, o funcionamento dos restantes sub módulos dependem também destas *flags*.

De seguida podemos ver a distribuição das *flags* de controlo neste registo.

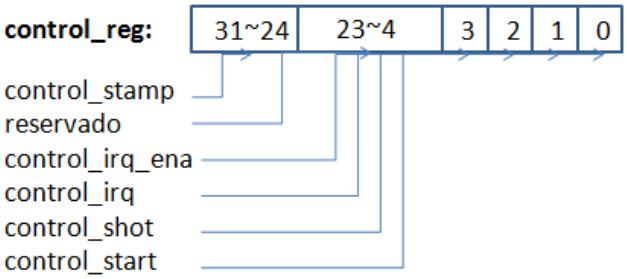

Figura 20 - Registo de controlo do módulo D5M

A *flag* **control_start** foi criada para "ancorar" a máquina de estados no estado **stoped** após *reset* do sistema. Esta *flag*, em conjugação com a palavra **control_stamp**, garantem que após *reset* o módulo fica no estado **stoped** até que a aplicação configure previamente os restantes registos de configuração. Pois tendo em conta de que este módulo envia a trama *RGB* para uma zona de memória do sistema é importante garantir que o registo **dma_address_reg** esteja devidamente configurado antes do arranque da máquina de estados.

No processo *VHDL* que controla a *flag* **control_start** podemos verificar que, após *reset* do sistema, esta *flag* é colocada a zero de forma assíncrona forçando a máquina de estados a "ancorar" no estado **stoped**. Esta estratégia funciona quando o botão de *reset* do sistema é ativado mas não garante que a máquina entre no estado **stoped** quando o sistema é ligado, pois o sinal de *reset* não é activo nestas situações, pelo que a palavra **control_stamp** garante que a máquina entre no estado **stoped** nestas situações.

A *flag* **control_shot** inibe a escrita no *fifo* quando o módulo está a funcionar no modo *snapshot*.

Para permitir o envio de pedidos de interrupção ao sistema sempre que uma trama *RGB* é enviada para o sistema a *flag* **control_irq_ena** deve ser ativa antes do arranque da máquina de estados ficando a *flag* **control_irq** reservada para efeitos de *acknowledge* por parte da rotina de atendimento da interrupção.

Apesar de o registo de controlo **control_reg** dar acesso de leitura e escrita, as *flags* que o constituem apenas podem ser configuradas quando a máquina de estados está no estado **stoped**. A título de exemplo, a *flag* **control_irq_ena** apenas pode ser configurada no estado **stoped** enquanto a *flag* **control_irq** apenas pode ser configurada no estado **rise_irq**.

No que diz respeito ao endereçamento, o sistema que contém uma interface *Avalon-MM Master*, endereça os registos ao *byte* no entanto, por mediação do *Interconnect Fabric*, do ponto de vista da interface *Slave* o endereçamento é feito à palavra (*word*). A título de exemplo, para aceder ao registo **frame_width_reg**, o *device driver* terá de aceder ao endereço base do controlador adicionando um *offset* de 0x4.

As especificações da interface *Avalon-MM Slave* podem ser consultadas no documento *Avalon Interface Specifications* em [16].

A máquina de estados do módulo de controlo é descrita em *VHDL* por intermédio de dois processos, síncrono e assíncrono. O processo síncrono **current_state** é responsável pela transição síncrona de estados da máquina em função do estado seguinte definido pelo processo assíncrono.

O processo assíncrono **next_state** gere a transição de estados em função das variáveis, entre as quais as *flags* de controlo, conforme nos sugere o *pseudo código* que se segue.

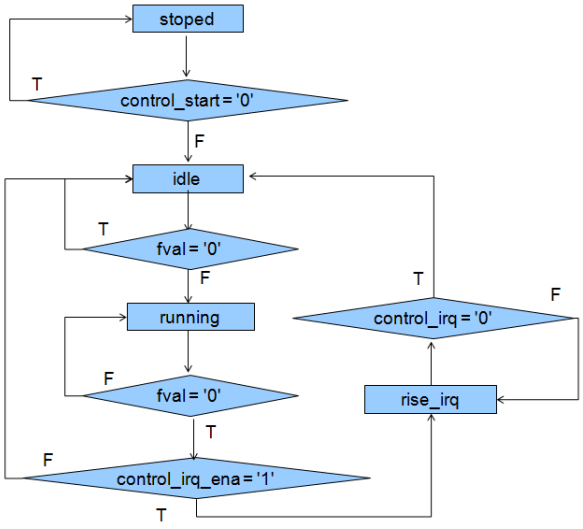

Figura 21 - Máquina de estados do módulo D5M

4.1.2 – Sub Módulo de Raw Data To RGB

Este módulo é responsável por receber os pixéis enviados pela câmara no formato *Bayer Pattern*, converter esta informação para o formato RGB e enviar para o *fifo* interno do módulo.

A câmara envia apenas uma componente *RGB* por cada ciclo de *clock*, conforme nos sugere a figura que se segue e este módulo terá de ter a capacidade de os agregar por forma a conseguir obter ambas as componentes por cada *pixel*.

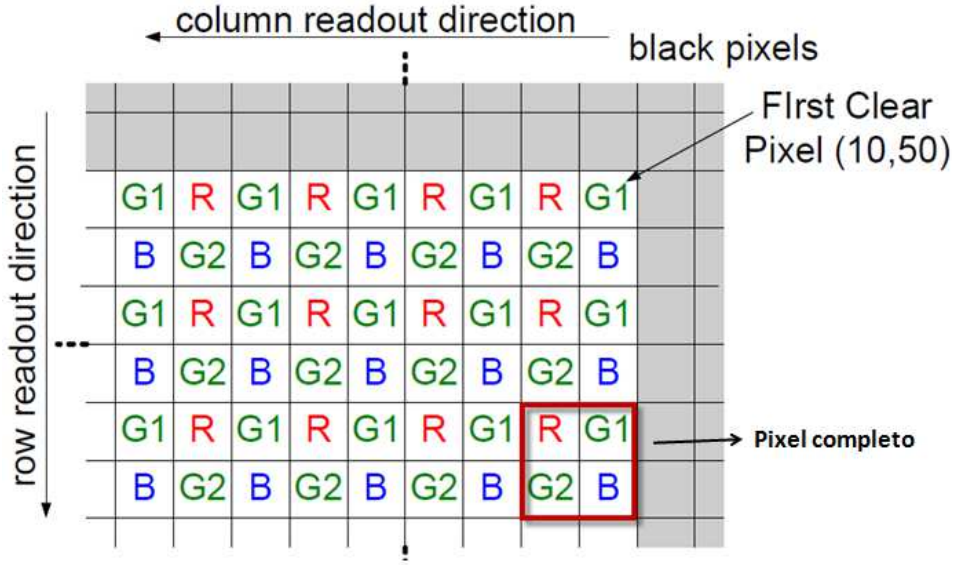

Figura 22 - Formato Bayer Pattern (Fonte: [22])

Podemos verificar, através da figura que se antecede, que os *pixéis* podem ser obtidos agregando duas colunas e duas linhas adjacentes pelo que se verifica a necessidade de

fazer um varrimento ao longo das linhas armazenando cada uma das componentes obtidas nas mesmas.

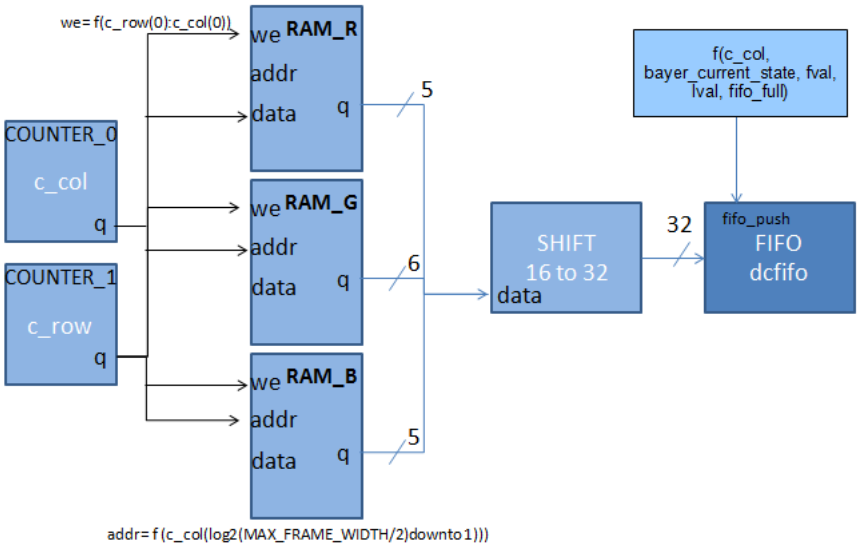

Figura 23 - Estrutura do Sub Modulo Raw Data To RGB

Para tal foi definido, em *VHDL*, uma entidade especializada para armazenar cada uma das componentes *RGB* ao longo das linhas. Esta entidade é escalonável ao nível da dimensão das palavras a armazenar (N: integer := 16) e ao nível do número de elementos a armazenar (M: integer:= 5) para que fosse facilmente adaptada ao formato *RGB16* onde as componentes não têm a mesma dimensão (número de *bits*).

No módulo são instanciadas entidades **bayer_ram** para cada componente *RGB* assignando ao sinal **we** a combinação lógica onde ocorrem as respetivas componentes e ao sinal **addr** o índice da coluna excluindo o *bit* de menor peso visto que, tal como já foi referido, cada componente *RGB* aparece no *bus* de dados da câmara (**data**) a cada dois ciclos de **pixclk**.
Note-se que as componentes *RGB* vão sendo concatenadas no sinal **c_pixel_data** ao ritmo de **pixclk**.
Uma vez que pretendemos enviar para o sistema palavras a 32 *bits* para minimizar o tempo de ocupação do *bus* os 16 *bits* resultantes de cada amostragem vão sendo concatenados no sinal **c_pixel_x0** através do *Shift Register* SHIFT_REG_0 e, desta forma, por cada ciclo de **pixclk** o sinal **c_pixel_x0** contem sempre a amostragem corrente e a anterior.

Verifique-se que as entidades **bayer_ram** e o *Shift Register* não dependem do estado atual do módulo de controlo, independentemente deste estado, estas entidades processam a informação meramente em função dos sinais **LVAL**, que valida os dados presentes no *bus* de dados da câmara e **pixclk**, para sincronismo.

O estágio final deste módulo corresponde a gerir o sinal **fifo_push**, que ativa a entrada *push* do *fifo* e a gestão do endereço de memória.

Tendo em conta de que teremos uma palavra (dois *pixéis*) por cada par de amostragens

basta incrementarmos o endereço de dois por cada ciclo **pixclk**.

O sinal **fifo_push** só vai permitir escritas no *fifo* quando a máquina de estados do módulo de controlo está no estado **running**.

O *fifo* guarda palavras a 64 *bits* porque, para além da palavra a 32 *bits* correspondente a duas amostragens, guarda o endereço de escrita.

4.1.3 – Sub Módulo Data

Este módulo é responsável por enviar as palavras que foram introduzidas no *fifo*, pelo módulo anterior, para o sistema **nios2**. Para o efeito disponibiliza ao sistema uma interface *Avalon-MM Master* para ser ligada ao dispositivo físico onde reside o *frame buffer*, neste caso o módulo *SRAM*.

O *bus address* está ligado aos 32 *bits MSB* da saída do *fifo* enquanto o *bus writedata* está ligado aos 32 *bits LSB* da saída do *fifo*. Desta forma apenas temos de gerir o sinal **m_write**, que está ligado ao sinal **write** da interface, e **fifo_pop**, que faz *pop* ao *fifo*.

Para implementar a interface *Avalon-MM Master* recorremos a uma máquina de estados que comuta entre dois estados em função do estado do *fifo*, reportado pelo sinal **fifo_empty**.

As palavras vão sendo retiradas do *fifo* sempre que o sinal **m_wait_request** estiver desativo, uma vez que este sinal é controlado pelo *slave* e ativo sempre que este não esteja disponível para atender ao pedido.

4.1.4 - Device Driver

Conforme verificamos no sub módulo de controlo, este dispõe de registos de configuração para adaptar o funcionamento do módulo às necessidades da aplicação e para o controlar, nomeadamente iniciar, parar, especificar o endereço do *frame buffer*, entre outros.

A implementação deste *device driver* é feita em três ficheiros distintos, nomeadamente *bayer_pattern_controller_regs.h*, *bayer_pattern_controller.h* e *bayer_pattern_controller.c*.

O *header bayer_pattern_controller_regs.h* contém o mapeamento dos registos de configuração e as macros com as rotinas básicas de acesso para cada um dos registos.

O *header bayer_pattern_controller.h* com a definição de estrutura deste dispositivo e a declaração das funções que fazem parte da *API* que o *device driver* disponibiliza à aplicação.

```
#ifndef _BAYER_PATTERN_CONTROLLER_H_
#define _BAYER_PATTERN_CONTROLLER_H_

#include "sys/alt_llist.h"
#include "priv/alt_dev_llist.h"
#include "priv/alt_file.h"
#include "bayer_pattern_controller_regs.h"

/*******************************************************************************
 * Bayer Pattern Private implementation
```

```
 ****************************************************************************************/
typedef void (*bayer_pattern_app_handler)(void *);
typedef struct bayer_pattern_pixel_struct
{
        alt_u8 r;
        alt_u8 g;
        alt_u8 b;
}bayer_pattern_pixel;
typedef struct bayer_pattern_struct
{
        alt_dev dev;
        int base_addr;
        //int * frame_buffer_24;
        //int * frame_buffer_16;
        bayer_pattern_app_handler irq_handler;

}bayer_pattern_dev;

#define BAYER_PATTERN_CONTROLLER_INSTANCE(name, dev)                                    \
static bayer_pattern_dev dev =                                          \
{                                       \
 {                                      \
    ALT_LLIST_ENTRY,                    \
    name##_NAME,                        \
    NULL, /* open */                    \
    NULL, /* close */                   \
    NULL, /* read */                    \
    NULL, /* write */                   \
    NULL, /* lseek */                   \
    NULL, /* fstat */                   \
    NULL, /* ioctl */                   \
 },                                     \
 name##_BASE,                           \
 NULL,                  \
}
static ALT_INLINE int bayer_pattern_device_register( alt_dev* fd)
{
   extern alt_llist bayer_pattern_dev_list;
   //Stop controller- force it to init state
   IOWR_BAYER_PATTERN_CONTROL_START(((bayer_pattern_dev*)fd)->base_addr, 0x0);
   return alt_dev_llist_insert ((alt_dev_llist*) fd, &bayer_pattern_dev_list);
}
#define BAYER_PATTERN_CONTROLLER_INIT(name, dev)        bayer_pattern_device_register((alt_dev*)&dev )
/****************************************************************************************
 * BAYER_PATTERN Public implementation
 ****************************************************************************************/
alt_dev* bayer_pattern_open_dev(const char* name);
int bayer_pattern_init_dev_no_interrupt(bayer_pattern_dev * dev, int fb, alt_u8 debug);
int bayer_pattern_init_dev(bayer_pattern_dev * dev, int fb, bayer_pattern_app_handler irq_handler, alt_u8 debug);
int bayer_pattern_max_frame_size(bayer_pattern_dev * dev);
#endif /* _BAYER_PATTERN_CONTROLLER_H_ */
```

Lista de Código 8 - Ficheiro bayer_pattern_controller.h

No que diz respeito à inicialização do módulo, este disponibiliza duas versões em função do facto de se pretender, ou não, ativar o pedido de interrupção sempre que exista uma trama *RGB* nova no *frame buffer*.

De seguida podemos visualizar a versão com suporte a pedido de interrupção sendo que a versão sem suporte de interrupção, consiste numa simplificação desta.

```
#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
 static void bayer_pattern_irq_handler(void * context)
#else
 static void bayer_pattern_irq_handler(void* context, alt_u32 id)
#endif
{
        //Cast context to bayer_pattern_dev *
        bayer_pattern_dev * dev_ptr = (bayer_pattern_dev *) context;
```

```
		if(dev_ptr->irq_handler != NULL)
			dev_ptr->irq_handler(context);
		//ack irq
		IOWR_BAYER_PATTERN_CONTROL_IRQ(dev_ptr->base_addr, 0x0);
}


int bayer_pattern_init_dev(bayer_pattern_dev * dev, int fb, bayer_pattern_app_handler irq_handler, alt_u8 debug)
{
	BAYER_PATTERN_DEBUG = debug;
	//Stop controller- force it to init state
	IOWR_BAYER_PATTERN_CONTROL(dev->base_addr, 0x0);
	dev->irq_handler = irq_handler;
	//config controller dma address
	IOWR_BAYER_PATTERN_DMA_ADDRESS(dev->base_addr, fb);
	IOWR_BAYER_PATTERN_DMA_MAX_SPAN(dev->base_addr, bayer_pattern_max_frame_size(dev));
	// Register the interrupt handler.
	#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
	alt_ic_isr_register(BAYER_PATTERN_0_IRQ_INTERRUPT_CONTROLLER_ID,BAYER_PATTERN_0_IRQ,
				bayer_pattern_irq_handler, (void*)dev,0x0);
	#else
	alt_irq_register( BAYER_PATTERN_0_IRQ, (void*)dev, bayer_pattern_irq_handler);
	#endif
	IOWR_BAYER_PATTERN_CONTROL(dev->base_addr, 0xAA000000);
	IOWR_BAYER_PATTERN_CONTROL_IRQ_ENA(dev->base_addr, 0x1);

	IOWR_BAYER_PATTERN_CONTROL_SHOT(dev->base_addr, 0x1);
	//Start controller - force it to idle state
	IOWR_BAYER_PATTERN_CONTROL_START(dev->base_addr, 0x1);
	return 0;
}
```

Lista de Código 9 - Ficheiro bayer_pattern_controller.c - bayer_pattern_init_dev

A função **bayer_pattern_init_dev** recebe, como argumentos, o endereço do *frame buffer* para onde o módulo Data vai escrever o conteúdo do *fifo* e o apontador para a função que vai atender a interrupção.

A inicialização do módulo passa por guardar o *handler* na estrutura e registá-lo no vetor de interrupções, configurar os registos **dma_address_reg**, **max_address_span_reg** e ativar a *flag* **control_irq_ena** no registo de controlo.

Por último, após salvaguardadas as configurações pretendidas, o módulo pode arrancar ativando a *flag* **control_start.**

A aplicação, para utilizar este recurso, deve recorrer à função **bayer_pattern_open_dev**, que retorna um apontador para o dispositivo e depois a função **bayer_pattern_init_dev**, ou **bayer_pattern_init_dev_no_interrupt** caso não pretenda recorrer às interrupções, começando o módulo a enviar tramas para o *frame buffer*.

4.2 - Bus I2C

Um *bus I2C* [21] é constituído por duas linhas, nomeadamente *SDA- Data Signal Line* e *SCL- Clock Signal Line* e implementa um protocolo de comunicação série em modo síncrono.

A linha *SCL* é responsável por transportar o sinal de sincronismo emitido pelo *master* que detém o controlo do *bus* no momento da transmissão, enquanto a linha *SDA* transporta os dados.

Do ponto de vista elétrico, ambas as linhas estão ligadas a um *pull up* pelo que quando estas estão em repouso, ou seja quando ambos os dispositivos estão com alta impedância, o valor lógico lido é "1".

O *I2C*, para além das características elétricas do *bus*, também especifica um protocolo com vista a sinalizar o início e fim da comunicação, bem como procedimentos para a aquisição do controlo do *bus* por parte de um dos *masters* e recuperação no caso de conflito na aquisição do controlo.

A interligação do *bus I2c* ao sistema **nios2** é feita recorrendo ao componente *PIO* **d5m_i2c**, instanciado no *SOPC Builder*, configurado conforme a figura que se segue.

Figura 24 - Componente PIO (Fonte: [6])

Como iremos ver, na implementação de *device driver*, para a implementação do protocolo de comunicação *I2C* ambas as linhas, *SDA* e *SCL*, terão de ser bidirecionais pelo que com esta configuração, o componente instanciado irá disponibilizar à entidade de topo apenas um *bus* para que esta ligue as linhas aos respetivos *pins* da placa.

A *flag* **Enable individual bit settinf/clearing** irá permitir efetuar *set* e *clear* sobre ambas as linhas, de forma individual, afetando os respetivos *bits* nos registos **outset** e **outclear** mapeados em memória para este componente.

A entidade de topo é responsável por instanciar o sistema **nios2** que, por sua vez, disponibiliza o porto com as duas linhas *SDA* e *SCL*. Estas linhas terão de ser ligadas ao porto *GPIO* da placa *DE2-115* que irá fornecer conectividade aos dispositivos *I2C* externos.

A correspondência de pinos entre o porto *GPIO* da placa e os pinos da *FPGA* pode ser vista na figura 35 (pode tirar a figura. Talvez passar para uma tabela em anexo ou colocar uma referência).

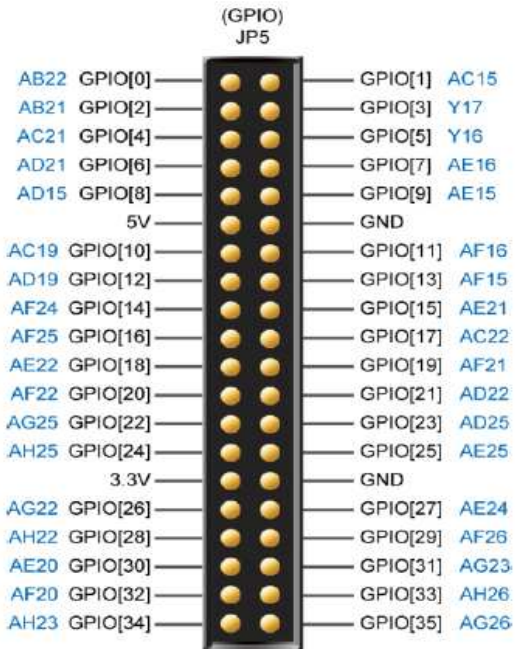

Figura 25 - Porto GPIO da placa DE-115 (Fonte: [4])

Desta forma iremos ligar a linha *SCL* ao porto *GPIO[24]* e a linha *SDA* ao porto *GPIO[23]*.

4.2.1 - Device Driver

O *device driver* tem a responsabilidade de interagir diretamente com o *hardware*, neste caso concreto o componente **d5m_i2c**, para implementar o protocolo de comunicação e fornecer uma *API* com as rotinas necessárias ao envio e receção de dados com total abstração ao *hardware*.

De seguida podemos visualizar o ficheiro *header i2c.h*.

```
#include "types.h"
#include "system.h"
#include "io.h"
/*********************************************************************************************
 * Architecture dependent part
 * If you change the architecture this is the only part that you should change
 * On this architecture I2C is acessible through PIO[scl,sda]
 *********************************************************************************************/
#define I2C_MAX_RECOVERS                        0x4
#define I2C_SCL_MAX_RATE                        100000
#define I2C_SCL_POS             0x0
#define I2C_SDA_POS                     0x1
#define I2C_SCL_MASK            (1 << I2C_SCL_POS)
#define I2C_SDA_MASK            (1 << I2C_SDA_POS)
/*
 * SDA related macros
 */
#define I2C_SDA_OUT()                   do \
                                        {       \
                                                u8 tmp = IORD_8DIRECT(D5M_I2C_BASE, 0x4);\
                                                tmp |= I2C_SDA_MASK;\
                                                IOWR_8DIRECT(D5M_I2C_BASE, 0x4, tmp);\
                                        }       \
                                        while(0)
```

```
#define I2C_SDA_IN()            do \
                                        {               \
                                                u8 tmp = IORD_8DIRECT(D5M_I2C_BASE, 0x4);\
                                                tmp &= (~I2C_SDA_MASK);\
                                                IOWR_8DIRECT(D5M_I2C_BASE, 0x4, tmp);\
                                        }               \
                                        while(0)
#define I2C_GET_SDA(data)               do      \
                                        {               \
                                                I2C_SDA_IN();\
                                                data = IORD_8DIRECT(D5M_I2C_BASE, 0x0);\
                                                data >>= I2C_SDA_POS;\
                                                data &= 0x1;\
                                        }               \
                                        while(0)
#define I2C_SET_SDA(data)               do      \
                                        {               \
                                                I2C_SDA_OUT();\
                                                if((data & 0x1) != 0)\
                                                    IOWR_8DIRECT(D5M_I2C_BASE, 0x10, I2C_SDA_MASK);\
                                                else\
                                                IOWR_8DIRECT(D5M_I2C_BASE, 0x14,  I2C_SDA_MASK);\
                                        }               \
                                        while(0)
/*
 * SCL related macros
 */
#define I2C_SCL_OUT()                   do              \
                                        {               \
                                                u8 tmp = IORD_8DIRECT(D5M_I2C_BASE, 0x4);\
                                                tmp |= I2C_SCL_MASK;\
                                                IOWR_8DIRECT(D5M_I2C_BASE, 0x4, tmp);\
                                        }               \
                                        while(0)
#define I2C_SCL_IN()            do              \
                                        {               \
                                                u8 tmp = IORD_8DIRECT(D5M_I2C_BASE, 0x4);\
                                                tmp &= (~I2C_SCL_MASK);\
                                                IOWR_8DIRECT(D5M_I2C_BASE, 0x4, tmp);\
                                        }               \
                                        while(0)
#define I2C_GET_SCL(data)               do      \
                                        {               \
                                                I2C_SCL_IN();\
                                                data = IORD_8DIRECT(D5M_I2C_BASE, 0x0);\
                                                data >>= I2C_SCL_POS;\
                                                data &= 0x1;\
                                        }               \
                                        while(0)
#define I2C_SET_SCL(data)               do      \
                                        {               \
                                                I2C_SCL_OUT();\
                                                if((data & 0x1) != 0)\
                                                IOWR_8DIRECT(D5M_I2C_BASE, 0x10, I2C_SCL_MASK);\
                                                else\
                                                        IOWR_8DIRECT(D5M_I2C_BASE, 0x14,  I2C_SCL_MASK);\
                                        }               \
                                        while(0)
#define I2C_PRE_INIT()          do              \
                                        {               \
                                                IOWR_8DIRECT(D5M_I2C_BASE, 0x10, IORD_8DIRECT(D5M_I2C_BASE,
0x10)& ~(I2C_SDA_MASK | I2C_SCL_MASK));\
                                                IOWR_8DIRECT(D5M_I2C_BASE, 0x14, IORD_8DIRECT(D5M_I2C_BASE,
0x14)& ~(I2C_SDA_MASK | I2C_SCL_MASK));\
                                                I2C_SDA_OUT();\
                                                I2C_SCL_OUT();\
                                                I2C_SET_SDA(0x1);\
                                                I2C_SET_SCL(0x1);\
                                        }               \
                                        while(0)
/*****************************************************************************
```

```
 * I2C Private implementation
 ********************************************************************************************/
#define I2C_STAUS_FREE                          1 << 0      //i2c line is free
#define I2C_STAUS_START_SENT            1 << 1      //success start condition was send
#define I2C_STAUS_ADDRESS_SENT          1 << 2      //address sent and ack by slave
#define I2C_STAUS_DATA_EXCHANGED        1 << 3      //data as been exchanged and ack
#define I2C_STAUS_ARBITRATION_LOST      1 << 4

typedef u32 (*i2c_timer_elapsed)(void );
/********************************************************************************************
 * I2C Public API
 ********************************************************************************************/
void i2c_init(i2c_timer_elapsed timer_elapsed, u32 timer_resolution, u8 debug);
u32 i2c_rcv(u8 address, u8* data, u8 n_bytes);
u32 i2c_send(u8 address, u8* data, u8 n_bytes, u8 stop);
```

Lista de Código 10 - Ficheiro i2c.h

A adaptação do *device driver* à arquitetura do sistema é feita através de *macros* que implementam as ações de escrita e leitura sobre as linhas *SDA* e *SCL*. Desta forma, para uma eventual reutilização do *device driver* basta redefinir as *macros*.

No caso concreto da arquitetura aqui em estudo, as *macros* recorrem ao *HAL* do componente *PIO* instanciado para implementar a ligação das linhas entre o sistema e os dispositivos *I2C*.

A *API* deste *device driver* disponibiliza as funções **i2c_init**, **i2c_rcv** e **i2c_send** que recebem o endereço do dispositivo *I2C* que se pretende alcançar.

A implementação do *device driver* é independente da arquitetura, uma vez que recorre às *macros* para aceder às linhas *SDA* e *SCL* e tem como objetivo implementar as funções da *API* obedecendo aos pressupostos de uma comunicação sobre um *bus I2C*.

4.3 - Módulo de controlo da Câmara

O módulo de controlo da Câmara irá ser implementado recorrendo ao *bus I2C* anteriormente desenvolvido, sendo o seu *device driver* responsável por recorrer ao *bus I2C* para configurar os registos internos da Câmara *D5M*.

O mapeamento dos registos, bem como o seu contexto, pode ser consultado no manual deste dispositivo em [22], pelo que aqui apenas nos iremos focar na implementação do *device driver*.

O ficheiro *header* deste device driver é constituído pelas macros de mapeamento dos registos internos da Câmara *D5M* e pela estrutura *d5m_dev* que encapsula o tipo *alt_dev*. Desta estrutura destacamos os atributos *D5M_ADDRESS* que contém o endereço *I2C* do módulo, *op_mode* para dar suporte ao modo contínuo e *snapshot* e *i2c_dev* que corresponde ao apontador para o *device driver I2C* que irá assegurar a comunicação série entre o sistema **nios2** e a Câmara *D5M*.

Com vista a obter uma total abstração ao facto de estarmos a recorrer ao *device driver do bus I2C* foram definidas as seguintes macros que permitem ler e escrever sobre determinado registo.

```
#define D5M_GET_REG(dev, reg, reg_value)do      \
                                        {       \
                                        alt_u8 reg_addr = reg;\
```

```
                                        if(i2c_send( dev->i2c_dev, dev->D5M_ADDRESS, &reg_addr, 1, 1) == 1)\
                                        {\
                                                alt_u8 tmp[2];\
                                                i2c_rcv(dev->i2c_dev, dev->D5M_ADDRESS, &tmp[0],
sizeof(tmp));\
                                                reg_value[0] = tmp[1];\
                                                reg_value[1] = tmp[0];\
                                                dev->D5M_STATUS = D5M_STAUS_OK;\
                                        }\
                                        else\
                                        {\
                                                dev->D5M_STATUS = D5M_STAUS_ERROR;\
                                        }\
                                }       \
                                while(0)
#define D5M_SET_REG(dev, reg, reg_value) do     \
                                {       \
                                        alt_u8 tmp[3];\
                                        tmp[0] = reg;\
                                        tmp[1] = reg_value[1];\
                                        tmp[2] = reg_value[0];\
                                        if(i2c_send(dev->i2c_dev,dev->D5M_ADDRESS, &tmp[0], sizeof(tmp), 1) ==
sizeof(tmp))\
                                        {\
                                                dev->D5M_STATUS = D5M_STAUS_OK;\
                                        }\
                                        else\
                                        {\
                                                dev->D5M_STATUS = D5M_STAUS_ERROR;\
                                        }\
                                }       \
                                while(0)
```

Lista de Código 11 - Ficheiro d5m.c - macros

A inicialização deste *device driver* é implementada na função **d5m_init_dev** que recebe o apontador para o dispositivo, previamente retornado pela função **d5m_open_dev**, e o apontador para o dispositivo *I2C*. Esta função inicializa os atributos relativo aos ganhos atuais das componentes, o modo de operação (contínuo ou *snapshot*).

```
void d5m_init_dev(d5m_dev* dev, i2c_dev* i2c_dev, alt_u8 i2c_address, alt_u8 debug)
{
        alt_u8 aux[2];
        dev->D5M_ADDRESS = i2c_address;
        dev->i2c_dev = i2c_dev;
        D5M_DEBUG = debug;
        //perform soft restart on D5M device
        d5m_restart(dev);
        /*
         * get current rgb gain
         */
        D5M_GET_REG(dev, D5M_REG_RED_GAIN,aux);
        dev->digital_r_gain = aux[1] & 0x7f;
        D5M_GET_REG(dev, D5M_REG_GREEN_1_GAIN,aux);
        dev->digital_g_gain = aux[1] & 0x7f;
        D5M_GET_REG(dev, D5M_REG_BLUE_GAIN,aux);
        dev->digital_b_gain = aux[1] & 0x7f;
        //Configure operation mode - continuos or snapshot
        D5M_GET_REG(dev, D5M_REG_READ_MODE_1,aux);
        switch(dev->op_mode)
        {
        case CONTINUOUS:
                aux[1] &= 0xfe;
                break;
        default://SNAPSHOT
                aux[1] |= 0x1;
        }
        D5M_SET_REG(dev, D5M_REG_READ_MODE_1,aux);
        //row and column read mode
```

```
        D5M_GET_REG(dev, D5M_REG_READ_MODE_2,aux);
        aux[1] |= (0x1 << 7);
        D5M_SET_REG(dev, D5M_REG_READ_MODE_2,aux);
        //LVAL, FVAL, and D[11:0] should be captured on the rising edge of PIXCLK
        aux[0] = 0x00;
        aux[1] = 0x80;
        D5M_SET_REG(dev, D5M_REG_PIXEL_CLOCK_CONTROL,aux);
        //perform default run time configuration
        d5m_config(dev, dev->D5M_RESOLUTION);
        //retrieve chip version
        D5M_GET_REG(dev, D5M_REG_CHIP_VERSION,dev->chip_version);
        d5m_logf("D5M Chip Version:0x%.2X%.2X detected.\n", \
                                dev->chip_version[1], dev->chip_version[0]);
}
```

Lista de Código 12 - Ficheiro d5m.c - d5m_init_dev

A câmara disponibiliza um campo de visão de 2592*1944 e tendo em conta de que não iremos implementar um sistema que suporte esta resolução porque a quantidade de transferências no *bus* de dados do sistema iria ser incomportável, o *device driver* irá permitir à aplicação a possibilidade de movimentar-se ao longo deste campo de visão, conforme nos sugere a figura que se segue.

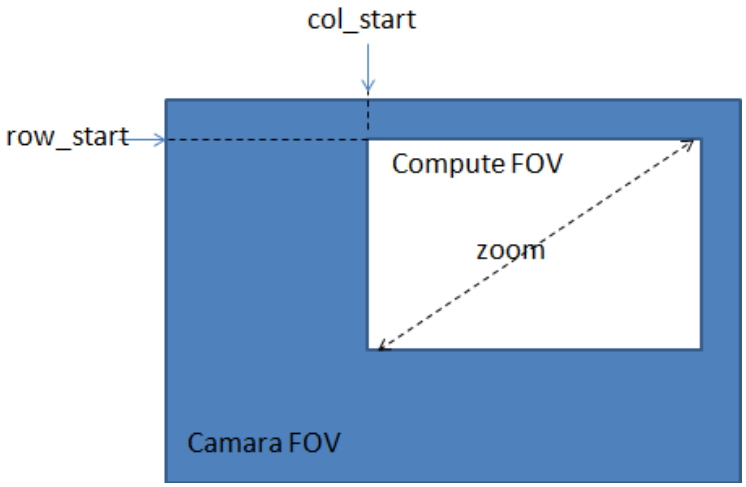

Figura 26 - Campo de visão da Câmara D5M

O *zoom* máximo é função da resolução adotada pelo sistema e do campo máximo de visão da câmara e pode ser determinado pela função **d5m_max_zoom**.

```
alt_u8 d5m_max_zoom(d5m_dev* dev)
{
        d5m_move(dev, 0, 0);
        alt_u8 max_zoom, i;
        switch(dev->D5M_RESOLUTION)
        {
        case RES_1024_768:
                max_zoom = (D5M_MAX_COL_SIZE/1024 < D5M_MAX_ROW_SIZE/768)? D5M_MAX_COL_SIZE/1024 :
D5M_MAX_ROW_SIZE/768;
                break;
        case RES_800_600:
                max_zoom = (D5M_MAX_COL_SIZE/800 < D5M_MAX_ROW_SIZE/600)? D5M_MAX_COL_SIZE/800 :
D5M_MAX_ROW_SIZE/600;
                break;
        case RES_640_480:
                max_zoom = (D5M_MAX_COL_SIZE/640 < D5M_MAX_ROW_SIZE/480)? D5M_MAX_COL_SIZE/640 :
D5M_MAX_ROW_SIZE/480;
                break;
```

```
		default://RES_1280_1024
			max_zoom = (D5M_MAX_COL_SIZE/1280 < D5M_MAX_ROW_SIZE/1024)? D5M_MAX_COL_SIZE/1280 :
D5M_MAX_ROW_SIZE/1024;
		}

		for(i = max_zoom; i % 2 != 0 && i > 0; i--){}

		return i;
}
```

Lista de Código 13 - Ficheiro d5m.c – d5m_max_zoom

A função **d5m_set_zoom** permite à aplicação alterar o *zoom*.

```
Void d5m_set_zoom(d5m_dev* dev, alt_u8 zoom)
{
		alt_u8 row_size_reg[2];
		alt_u8 col_size_reg[2];
		alt_u8 row_start_reg[2];
		alt_u8 col_start_reg[2];
		alt_u8 row_bin_reg[2];
		alt_u8 col_bin_reg[2];
		//alt_u8 max_zoom;

		if(zoom <= d5m_max_zoom(dev) && zoom % 2 == 0)
		{
			dev->zoom = zoom;
		}
		else
		{
			dev->zoom = 1;
		}

		D5M_GET_REG(dev, D5M_REG_COLUMN_ADDRESS_MODE,col_bin_reg);
		col_bin_reg[0] &= 0xf8;

		D5M_GET_REG(dev, D5M_REG_ROW_ADDRESS_MODE,row_bin_reg);
		row_bin_reg[0] &= 0xf8;

		switch(dev->D5M_RESOLUTION)
		{
		case RES_1024_768:
			break;
		case RES_800_600:
			col_size_reg[0] = ((dev->zoom * 800)- 1) & 0xff;
			col_size_reg[1] = (((dev->zoom * 800)- 1) >> 8) & 0xff;
			row_size_reg[0] = ((dev->zoom * 600)- 1) & 0xff;
			row_size_reg[1] = (((dev->zoom * 600)- 1) >> 8) & 0xff;

			col_start_reg[0] = ((D5M_MAX_COL_SIZE – 800)/(2 * dev->zoom)) & 0xff;
			col_start_reg[1] = (((D5M_MAX_COL_SIZE – 800)/(2 * dev->zoom)) >> 8 ) & 0xff;

			row_start_reg[0] = ((D5M_MAX_ROW_SIZE – 600)/(2 * dev->zoom)) & 0xff;
			row_start_reg[1] = (((D5M_MAX_ROW_SIZE – 600)/(2 * dev->zoom)) >> 8 ) & 0xff;

			col_bin_reg[0] |= (dev->zoom – 1) & 0x7;
			row_bin_reg[0] |= (dev->zoom – 1) & 0x7;
			break;
		case RES_640_480:
			break;
		default://RES_1280_1024
			break;
		}
		//set values
		D5M_SET_REG(dev, D5M_REG_ROW_SIZE,row_size_reg);
		D5M_SET_REG(dev, D5M_REG_COLUMN_SIZE,col_size_reg);
		D5M_SET_REG(dev, D5M_REG_ROW_START,row_start_reg);
		D5M_SET_REG(dev, D5M_REG_COL_START,col_start_reg);
		D5M_SET_REG(dev, D5M_REG_ROW_ADDRESS_MODE,row_bin_reg);
		D5M_SET_REG(dev, D5M_REG_COLUMN_ADDRESS_MODE,col_bin_reg);
}
```

Lista de Código 14 - Ficheiro d5m.c - d5m_set_zoom

No que diz respeito ao posicionamento, a função **d5m_move** calcula o posicionamento inicial das colunas e linhas em função do *zoom* atual e da posição pretendida. Note-se que esta função retorna < 0 quando a posição pretendida não é alcançável para que a aplicação, ou mesmo o utilizador, tenha conhecimento.

```
int d5m_move(d5m_dev* dev, int x, int y)
{
	int row_start, col_start, col_size, row_size, zoom;
	alt_u8 row_start_reg[2];
	alt_u8 col_start_reg[2];
	alt_u8 col_bin_reg[2];

	D5M_GET_REG(dev, D5M_REG_COLUMN_ADDRESS_MODE,col_bin_reg);
	zoom = dev->zoom;//(col_bin_reg[0] & 0x7) + 1;

	switch(dev->D5M_RESOLUTION)
	{
	case RES_1024_768:
		break;
	case RES_800_600:
		col_size = zoom * 800;
		row_size = zoom * 600;
		col_start = ((D5M_MAX_COL_SIZE - 800)/(2 * zoom)) + x * zoom;
		row_start = ((D5M_MAX_ROW_SIZE - 600)/(2 * zoom)) + y * zoom;
		break;
	case RES_640_480:
		break;
	default://RES_1280_1024
		break;
	}
	//set values
	row_start_reg[0] = row_start & 0xff;
	row_start_reg[1] = (row_start >> 8) & 0xff;

	col_start_reg[0] = col_start & 0xff;
	col_start_reg[1] = (col_start >> 8) & 0xff;

	if(row_start > 0 && row_start < D5M_MAX_ROW_SIZE - row_size)
	{
		if(col_start > 0 && col_start < D5M_MAX_ROW_SIZE - col_size)
		{
			D5M_SET_REG(dev, D5M_REG_COL_START,col_start_reg);
			D5M_SET_REG(dev, D5M_REG_ROW_START,row_start_reg);
			dev->center_x = x;
			dev->center_y = y;
			return 0;
		}
	}
	return -1;
}
```

Lista de Código 15 - Ficheiro d5m.c - d5m_move

4.4 - Compressão de Imagem

O envio de tramas RGB para a rede Ethernet torna- se pouco eficiente uma vez que uma trama com resolução de 800*600 contém 800*600*2 bytes tornando- se mesmo incomportável se considerarmos que a aplicação cliente está a correr num dispositivo móvel.

Para implementação da compressão das imagens captadas pela Câmara para o formato JPEG recorremos à biblioteca libjpeg [26] que consiste num pacote de *software* aberto com funcionalidade de compressão e descompressão.

4.4.1 – Adaptação da Biblioteca

Adicionalmente ao código aberto, a biblioteca libjpeg disponibiliza documentação acerca da sua utilização. Desta documentação, podemos verificar que a sua utilização passa essencialmente pela instanciação de um objeto **jpeg_compress_struct** que contém os atributos necessários com os parâmetros de compressão pretendidos.

O objecto **jpeg_compress_struct** contém, como atributo, um apontador para o objeto **jpeg_destination_mgr** que permite redefinir a forma como a biblioteca escreve no ficheiro de destino. Esta adaptação é feita redefinindo os três *handlers* que este objeto contém conforme nos sugere o excerto de código que se segue.

```
//Initialization takes place in the callback mem_init_destination():
static void jpeg_init_destination(j_compress_ptr cinfo)
{
	cinfo->dest->next_output_byte = (JOCTET*)&jpeg_fb_ptr[0];
	memcpy(cinfo->dest->next_output_byte, &mjpeg_header[0], sizeof(mjpeg_header) - 1);
	cinfo->dest->next_output_byte += sizeof(mjpeg_header) - 1;
	cinfo->dest->free_in_buffer = JPEG_FILE_SIZE - sizeof(mjpeg_header) + 1;
	libjpeg_time_stamp = alt_nticks();
}
//When the buffer size is not large enough, the library requests more data in the following callback:
static boolean jpeg_empty_output_buffer(j_compress_ptr cinfo)
{
	printf("no enough space.\n");
	return 0;
}
//When the compression has finished, we need to resize the buffer to the actual size:
static void jpeg_term_destination(j_compress_ptr cinfo)
{
	libjpeg_file_size = (int)(jpeg_mgr.next_output_byte - 1 - ((int)&jpeg_fb_ptr[0]));
	libjpeg_time_stamp = (alt_nticks() - libjpeg_time_stamp)/alt_ticks_per_second();
	fprintf(stderr, "jpeg file write completed. %u bytes written in %u seconds.\n", \
				libjpeg_file_size, libjpeg_time_stamp);
}
static void jpeg_set_mem_dest( j_compress_ptr cinfo, struct jpeg_destination_mgr * dst )
{
	cinfo->dest = dst;
	cinfo->dest->init_destination = jpeg_init_destination;
	cinfo->dest->term_destination = jpeg_term_destination;
	cinfo->dest->empty_output_buffer = jpeg_empty_output_buffer;
}
```

Lista de Código 16 - Ficheiro libjpeg.c - Adaptação da Biblioteca

A função **jpeg_init_destination** é responsável por inicializar o apontador para o *Frame Buffer JPEG*, para onde se pretende enviar a imagem comprimida, e definir o número de *bytes* disponíveis bem como inicializar o *time stamp* para um eventual controlo do ritmo de compressão.

A função **jpeg_empty_output_buffer** é evocada pela biblioteca sempre que a memória alocada não seja suficiente.

Por último, a função **jpeg_term_destination** é evocada pela biblioteca quando esta termina a compressão e onde são calculados o *time stamp* e a dimensão do ficheiro

gerado.

Uma vez que a compressão para *JPEG* é um processo recursivo para o processador do sistema verifica- se a necessidade de implementar uma forma de libertar o processador para que esta esteja disponível para as restantes funções. Pois, por exemplo, caso o utilizador pretenda aceder à interface *http* da Câmara e se estiver a decorrer uma compressão de imagem o sistema terá de ter a capacidade de atender a este pedido.

Para este efeito a compressão é feita através de várias chamadas à função **rgb_2_jpeg_compress** que apenas retorna zero quando todas as linhas da imagem forem passadas para a biblioteca.

```
int rgb_2_jpeg_compress(void)
{
        if(c_row == 0)
                goto stage0;
        else if(c_row < IMAGE_HEIGHT)
                goto stage1;
        else
                goto stage2;
stage0:
         /*set up the error handler first, in case the initialization*/
         cinfo.err = jpeg_std_error(&jerr);
         /* Now we can initialize the JPEG compression object. */
         jpeg_create_compress(&cinfo);
         /*set parameters for compression */
         cinfo.image_width = IMAGE_WIDTH;
         cinfo.image_height = IMAGE_HEIGHT;
         cinfo.input_components = 3;
         cinfo.in_color_space = JCS_RGB;
         /* Now use the library's routine to set default compression parameters.*/
         jpeg_set_defaults(&cinfo);
         /*set any non-default parameters you wish to.*/
         jpeg_simple_progression (&cinfo);
         jpeg_set_quality(&cinfo, 80, TRUE);
         cinfo.dct_method = JDCT_IFAST;
         /*specify data destination */
         jpeg_set_mem_dest( &cinfo, &jpeg_mgr);
         /*Start compressor */
         jpeg_start_compress(&cinfo, TRUE);
stage1:
         while (cinfo.next_scanline < IMAGE_HEIGHT)
         {
                 rgb_row_depth_16_to_24((int *)&row_buffer[0], \
                                 (int*)&image_buffer[cinfo.next_scanline * 2 * \
                                 IMAGE_WIDTH ] , IMAGE_WIDTH/2);
                 jpeg_write_scanlines(&cinfo, row_pointer, 1);
                 ++c_row;
                 goto stage3;
         }
stage2:
         /*################Step 6: Finish compression#################*/
         if (cinfo.global_state == CSTATE_SCANNING ||
            cinfo.global_state == CSTATE_RAW_OK) {
          /* Terminate first pass */
```

```
            if (cinfo.next_scanline < cinfo.image_height)
              ERREXIT(&cinfo, JERR_TOO_LITTLE_DATA);
            (*cinfo.master->finish_pass) (&cinfo);
          } else if (cinfo.global_state != CSTATE_WRCOEFS)
            ERREXIT1(&cinfo, JERR_BAD_STATE, cinfo.global_state);
          /* Perform any remaining passes */
          while (! cinfo.master->is_last_pass)
          {
            (*cinfo.master->prepare_for_pass) (&cinfo);
            for (iMCU_row = 0; iMCU_row < cinfo.total_iMCU_rows; iMCU_row++) {
              if (cinfo.progress != NULL) {
                    cinfo.progress->pass_counter = (long) iMCU_row;
                    cinfo.progress->pass_limit = (long) cinfo.total_iMCU_rows;
                    (*cinfo.progress->progress_monitor) ((j_common_ptr) &cinfo);
              }
              /* We bypass the main controller and invoke coef controller directly;
               * all work is being done from the coefficient buffer.
               */
              if (! (*cinfo.coef->compress_data) (&cinfo, (JSAMPIMAGE) NULL))
                    ERREXIT(&cinfo, JERR_CANT_SUSPEND);
            }
            (*cinfo.master->finish_pass) (&cinfo);
          }
          /* Write EOI, do final cleanup */
          (*cinfo.marker->write_file_trailer) (&cinfo);
          (*cinfo.dest->term_destination) (&cinfo);
          /* We can use jpeg_abort to release memory and reset global_state */
          jpeg_abort((j_common_ptr) &cinfo);
          /*release JPEG compression object */
          jpeg_destroy_compress(&cinfo);
          /* And we're done! */
          return 0;
stage3:
          return  1; }
```

Lista de Código 17 - Ficheiro libjpeg.c - Compressão

O inteiro **c_row** serve para controlar o número de vezes que a aplicação chama a função e, em função deste, determina para que zona do código deve passar.

Capítulo 5 – Desenvolvimento Aplicacional

Após implementação da arquitetura pretendida, o desenvolvimento aplicacional irá controlar os módulos para que o sistema cumpra os pressupostos pretendidos.

Alguns destes módulos são totalmente independentes da aplicação sendo apenas necessário configurar os seus registos internos, para que estes funcionem conforme pretendido, e dar- lhes a indicação para que começam a processar.

Para além do controlo dos módulos do sistema, verifica- se a necessidade de implementar a interação entre o sistema e o utilizador que é feita através de ligações *http* e para tal será necessário o desenvolvimento de uma aplicação servidora capaz de aceitar pedidos neste protocolo e retribuir o conteúdo requisitado.

5.1 - Servidor Web

Neste tópico iremos abordar a implementação de um servidor *web* que recorre à interface *Socket* implementada no Capítulo 3.

A implementação da interface *Socket* dispõe de um *buffer* para manter os segmentos que são enviados da aplicação e como tal, esta terá de lidar com situações de *buffer overflow* na eventualidade de estar a debitar informação a um ritmo muito superior ao fluxo da rede.

A plataforma em uso não dispõe de mecanismos de paralelismo como, por exemplo, *threads* e processo. Tal leva a um cuidado acrescido com as funções bloqueantes como por exemplo a aplicação ficar presa num *loop* à espera que o *buffer* fique com espaço disponível, e como o processador não é libertado, o *socket* nunca irá enviar as tramas para a rede.

Para ultrapassar ambas as situações a implementação adotada recorre a uma implementação de *threads*, novamente da autoria do Sr. Adam Dunkels, que consiste num mecanismo de programação concorrente vulgarmente utilizado em sistemas embebidos com recursos de memória muito limitados. Pois nesta implementação a *thread* não dispõe de *stack* dedicado e a sua utilização consiste no recurso a *macros* definidas no *header pt.h*.

A aplicação servidora aqui implementada responde a pedidos *http*, em função do *uri* enviado pelo cliente, e responde através da interface *Socket* conteúdo pretendido em linguagem *html*.

De seguida podemos visualizar a estrutura adotada para dar suporte à implementação desta aplicação servidora, nomeadamente o *header sys-monitor.h*.

```
#ifndef SYS_MONITOR_
#define SYS_MONITOR_

#include "net.h"
#include "pt-sem.h"
```

```
#define WWW_TCP_PORT                    80
#define WWW_MAX_CONNECTION              10 //we handle 4 paralell connections

typedef char monitor_handler(void *);
typedef volatile struct monitor_state_struct
{
        monitor_handler * handler;
        socket * sock;
        struct pt_sem mutex;

        char get[256];
        char * p;
        char * uri;
        char * formdata;
        char line[40];
        char * formlist[10];
        /*********************************************************************
                A Protothread to implement paralell connection
        *********************************************************************/
        //Protothreads to handle request/response
        struct pt request, response;//, body_begin, body_end;
        //u16_t sent_size; //holds the size of sent bytes from the last block. it will be
        u32_t sent_size; //holds the size of sent bytes from the last block. it will be
        int result;
} monitor_state;
typedef volatile struct __attribute__ ((aligned)) monitor_table_entry_struct
{
   monitor_handler * handler;
   char             * pattern;
} monitor_table_entry;

/*********************************************************************
        Monitor API
*********************************************************************/
void monitor_init(void);
void monitor_schedule(void);
#endif /* SYS_MONITOR_ */
```

Lista de Código 18 - Ficheiro sys-monitor.h

A implementação do servidor *web*, que consta no ficheiro fonte *sys-monitor.c*, dispõe de um *array* de estruturas do tipo ***monitor_state*** e, como podemos verificar no *header*, dispõe de diversos atributos para permitir o controlo necessário ao atendimento e manutenção das diversas sessões de *http*. Entre os quais se destacam o *handler* responsável por processar a resposta ao método *GET*, um apontador para o *socket* retornado pela *API* na função ***net_server_accept***, um *Semaphore* para implementar sincronismo entre *threads* e duas *Protothreads* que implementam o processamento dos pedidos e respetivas respostas.

Adicionalmente, para além das duas *threads* por ligação, existe uma terceira *thread* declarada como variável global ***main_thread*** que implementa a função que efetua o atendimento das ligações.

A função ***monitor_schedule*** é responsável por implementar o paralelismo no atendimento das ligações e processamento das respetivas respostas. Esta recorre à

macro ***PT_SCHEDULE*** que evoca a *thread* que lhe é passada como argumento. Por sua vez, a *thread* evocada, ou uma *child* desta, retornará logo que seja evocada uma espera passiva ou através da macro ***PT_END*** que sinaliza o fim da execução da *thread*. Cada vez que uma função ***monitor_schedule*** é evocada lança a *thread* ***monitor_handle_connection*** e as *threads* ***monitor_handle_request*** referentes a cada uma das ***monitor_state*** do *array* ***monitor_stack***.

A *thread* ***monitor_handle_connection***, fica em espera passiva por uma estrutura ***monitor_state*** livre no *array* e seguidamente, por um pedido de estabelecimento de ligação através da função ***net_server_accept*** da *API Socket*. Em ambos os casos a espera passiva é feita recorrendo à *macro* ***PT_WAIT_WHILE*** que retorna imediatamente para a função ***monitor_schedule*** sempre que o seu argumento for verdadeiro. Logo que seja retornado um *socket* válido da *API* o semáforo ***mutex*** é sinalizado através da *macro* ***PT_SEM_SIGNAL*** para que a *thread* ***monitor_handle_request*** dê início ao processamento da respetiva ligação.

A *thread* ***monitor_handle_request*****,** depois de sinalizado o semáforo, lê o *buffer* de receção do *socket* até encontrar o método *GET* e respetivo *uri* para determinar qual o *haldler* a utilizar para processar e enviar a resposta para o *peer* remoto.

```
static PT_THREAD(monitor_handle_request(monitor_state * s) )
{
	static s32_t result;
	static monitor_table_entry * entry;
	PT_BEGIN(&s->request);
	while(1)
	{
		//passively waits for a connection request from remote host (http client)
		PT_SEM_WAIT(&s->request, &s->mutex);
		assert(s->sock != NULL && s->handler == NULL);
		//whait for http get header
		PT_WAIT_WHILE(&s->request, (result = monitor_getline(s->sock, (char*)&s->get[0],
sizeof(s->get))) == 0);
		if(result == SOCKET_ERROR)
		{
			monitor_logf("[monitor_handle (%p)]->socket error!!!\r\n", s->sock);
			goto close;
		}
		monitor_logf("[monitor_handle (%p)]->Request <%s>\r\n", s->sock, s->get);

		//waits for blanc line as defined on http standard
		PT_WAIT_WHILE(&s->request, (result = monitor_getline(s->sock, (char*)&s->line[0],
sizeof(s->line))) > 0);
		if(result == SOCKET_ERROR){	goto close;}

		//process request
		s->formdata = NULL;
		s->uri = s->p = (char*)&s->get[0] + 4;
		while( *s->p != ' ' && *s->p != '?' )
			s->p++;
		if( *s->p == '?' )
			s->formdata = s->p + 1;
		*s->p = 0;
		if (s->formdata != NULL )
```

```
                {
                        while( *s->p != ' ' )
                                s->p++;
                        *s->p = 0;
                }
                monitor_logf("[monitor_handle (%p)]->Request filename >%s< formdata >%s<\r\n", s-
>sock, s->uri, \
                                s->formdata ? s->formdata: "-NULL-");
                entry = &http_table[0];
                while(entry->handler != NULL)
                {
                        if (match(s->uri, entry->pattern)){  break;}
                        ++entry;
                }
                if (entry->handler != NULL)
                {
                        s->handler = entry->handler;
                        assert(s->handler != NULL);
                        monitor_logf("[monitor_handle (%p)]->calling %p: %s\r\n", s->sock, entry,
entry->pattern);
                        PT_SPAWN(&s->request, &s->response, s->handler((void *)s));
                }
                else
                {
                        goto close;
                }
                close:
                //Close the Socket
                monitor_logf("[monitor_handle (%p)]->%s\r\n", s->sock, "Closing connection");
                net_close(s->sock);
                s->sock = NULL;
                s->handler = NULL;
        }
        PT_END(&s->request);
}
```

Lista de Código 19 - Ficheiro sys-monitor.c - monitor_handle_request

As *child threads*, responsáveis por processar as respostas, são evocadas através da *macro* ***PT_SPAWN*** na *thread* ***monitor_handle_request*** e recorrem às *macros* ***WAIT_FOR_STREAM*** e ***WAIT_FOR_STREAM_F*** para enviar o código *html* para o cliente.

Para lidar com as situações de *buffer overflow* foi implementada a função especializada ***monitor_send*** que apenas coloca zero no atributo ***sent_size*** quando a totalidade de informação a enviar é transferida para o *buffer*. Caso contrário, compete à função que a evoca verificar o atributo e evocá-la até que este chegue a zero. Adicionalmente, as *macros* ***WAIT_FOR_STREAM*** e ***WAIT_FOR_STREAM_F*** foram definidas para encapsular a função ***monitor_send*** e libertar o processador, pois de outra forma o conteúdo *buffer* nunca seria transferido para a rede.

5.1.1 – Suporte para streaming

A aplicação cliente toma conhecimento do tipo de conteúdo através do envio, no campo **Content-Type**, do tipo *MIME*, *multipart/x-mixed-replace;boundary=jpeg-streamer*.

A imagem captada pela Câmara é enviada para o cliente no formato *MJPEG* [25], que consiste no envio sucessivo de tramas *JPEG* delimitadas pela *string* especificada no campo *boundary*.

A *thread* **monitor_stream** fica em espera passiva até que a função **rgb_2_jpeg_compress** retorne zero que, conforme mencionado no Capítulo 5, sinaliza o fim da compressão de uma imagem e recorre à *macro* ***WAIT_FOR_STREAM*** para enviar a mesma para a aplicação cliente.

```
static PT_THREAD(monitor_stream(void * s))
{
	PT_BEGIN(&((monitor_state *)s)->response);
	WAIT_FOR_STREAM_F(((monitor_state *)s), \
				"%s\r\n%s\r\n%s\r\n%s\r\n%s\r\n%s\r\n", \
				"HTTP/1.1 200 OK", \
				"Content-Type: multipart/x-mixed-replace;boundary=jpeg-streamer", \
				"Cache-Control: no-store", \
				"Pragma: no-cache",\
				"Connection: close\r\n", "\r\n");
	while(((monitor_state *)s)->result >= 0)
	{
		//passively waits for compressed data on the file.
		PT_WAIT_WHILE(&((monitor_state *)s)->response, \
		jpeg_dst_mgr.dma->read_addr == jpeg_dst_mgr.dma->write_addr);
		jpeg_n_bytes = (jpeg_dst_mgr.dma->write_addr - jpeg_dst_mgr.dma->read_addr >=
128)? 128 : jpeg_dst_mgr.dma->write_addr - jpeg_dst_mgr.dma->read_addr;
		WAIT_FOR_STREAM(((monitor_state *)s), (char*)jpeg_dst_mgr.dma->read_addr,
jpeg_n_bytes);
		jpeg_dst_mgr.dma->read_addr += jpeg_n_bytes;
		//We must free processor to handle other stuff, namely sending data to the network
through socket.
		PT_YIELD(&((monitor_state *)s)->response);
	}
	jpeg_dst_mgr.dma->read_addr = jpeg_dst_mgr.dma->write_addr;

	PT_END(&((monitor_state *)s)->response);
}
```

Lista de Código 20 - Ficheiro sys-monitor.c - monitor_stream

5.2 - Aplicação

Ao nível da aplicação, o sistema é gerido através dos objetos instanciados que, por sua vez, acedem direta ou indiretamente aos módulos do sistema através dos respetivos *device drivers*.

A figura que se segue sugere a forma como estes objetos foram instanciados bem como a iteração que estes têm com o *hardware* do sistema.

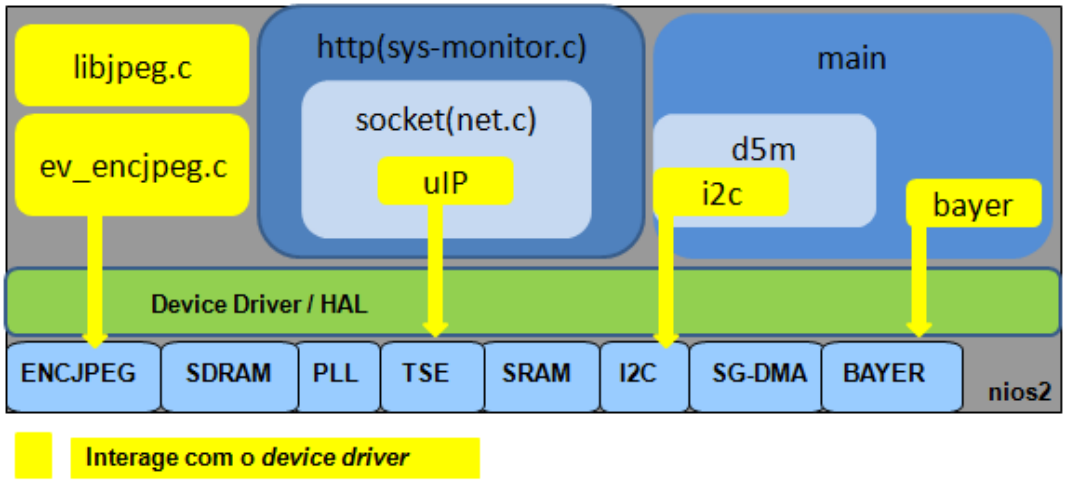

Figura 27 – Aplicação

O objeto *rgb2jpeg*, responsável pela gestão do módulo *ENCJPEG*, é configurado em tempo de compilação em termos do número de *frame buffers* que pretendemos e da dimensão máxima de memória alocada para o efeito ficando a função *main* apenas responsável por evocar a função *rgb_2_jpeg_schedule* para que o objeto faça uma gestão dos *frame buffers* que deverão ser escritos e lidos.

A Câmara *D5M*, conforme explicado no Capítulo 4, é gerida pelo objeto *d5m* ao nível da função *main* sendo apenas necessário inicializar o dispositivo em modo contínuo, através do seu *device* driver recorrendo à *função d5m_init_dev* passando um apontador para o dispositivo *i2c* que irá assegurar a comunicação série com a Câmara.

A Câmara *D5M*,também necessita do objeto *bayer* para assegurar a conversão dos pixéis do formato *bayer pattern* para *RGB*. Este objeto também é inicializado na função *main* passando o endereço de memória onde reside o *frame buffer RGB*, através da função *bayer_pattern_init_dev_no_interrupt*.

```
int main()
{
  printf("+----------------------------------------+\n");
  printf("| System initialization...                |\n");
  printf("+----------------------------------------+\n");
  while(!ALTPLL_LOCKED);
  /*****************************************************************************
   * I2c Initialization
   *****************************************************************************/
  printf(" - Initializing I2C controller...\n");
  i2c = (i2c_dev *)i2c_open_dev("/dev/i2c");
  i2c_init_dev(i2c, alt_nticks, alt_ticks_per_second(), 0);
  printf("   > device name:%s.\n", i2c->dev.name);
  printf("   > logical address:%p.\n", i2c);
  printf("   > physical address:0x%x.\n", i2c->base_addr);
  if(i2c == NULL)
  {
          goto app_exit;
  }
```

```
/*****************************************************************************
 * D5M Initialization
 *****************************************************************************/
printf(" - Initializing D5M controller...\n");
d5m = (d5m_dev*)d5m_open_dev("/dev/d5m");
//d5m->op_mode= SNAPSHOT;
d5m_init_dev(d5m, i2c, 0x5d, 0);
printf("   > device name:%s.\n", d5m->dev.name);
printf("   > logical address:%p.\n", d5m);
printf("   > i2c address:0x%x.\n", d5m->D5M_ADDRESS);
printf("   > associated I2C controller [%p].\n", d5m->i2c_dev);
if(d5m == NULL)
{
        goto app_exit;
}
/*****************************************************************************
 * BAYER PATTERN Initialization
 *****************************************************************************/
printf(" - Initializing BAYER PATTERN controller...\n");
bayer = (bayer_pattern_dev*)bayer_pattern_open_dev("/dev/bayer_pattern_0");
bayer_pattern_init_dev_no_interrupt(bayer, RGB_FRAME_BUFFER_ADDRESS, 0);
printf("   > device name:%s.\n", bayer->dev.name);
printf("   > logical address:%p.\n", bayer);
printf("   > physical address:0x%x.\n", bayer->base_addr);
printf("   > current frame buffer:%x.\n", IORD_BAYER_PATTERN_DMA_ADDRESS(bayer->base_addr));
printf("   > max. resolution:%u*%u*%u.\n", \
                IORD_BAYER_PATTERN_MAX_FRAME_WIDTH(bayer->base_addr), \
                IORD_BAYER_PATTERN_MAX_FRAME_HEIGHT(bayer->base_addr), \
                IORD_BAYER_PATTERN_COLOR_DEPTH(bayer->base_addr));
printf("   > frame size:%d [bytes]\n", IORD_BAYER_PATTERN_DMA_MAX_SPAN(bayer->base_addr));
if(bayer == NULL)
{
        goto app_exit;
}
/*****************************************************************************
 * jpeg encoder Controller Initialization
 *****************************************************************************/
        printf(" - Initializing jpeg encoder controller...\n");
        jpeg = jpegenc_open_dev("/dev/encjpeg_0");
        if(jpeg == NULL)
        {
                goto app_exit;
        }
        jpeg_fb_ptr = (char *)JPEG_FRAME_BUFFER_ADDRESS;//&jpeg_fb[0];
        IOWR_ENCJPEG_RGB_FB_ADDR_REG(jpeg->base_addr, RGB_FRAME_BUFFER_ADDRESS);
        IOWR_ENCJPEG_JPEG_FB_ADDR_SPAM_REG(jpeg->base_addr, JPEG_FILE_SIZE);
        IOWR_ENCJPEG_JPEG_FB_ADDR_REG(jpeg->base_addr, (int)jpeg_fb_ptr);
        jpegenc_config(jpeg, 640, 480);
        printf("   > device name:%s.\n", jpeg->dev.name);
        printf("   > resolution:%u*%u.\n", IORD_ENCJPEG_IMAGE_WIDTH(jpeg->base_addr),
IORD_ENCJPEG_IMAGE_HEIGHT(jpeg->base_addr));
        printf("   > rgb frame buffer:%x.\n", IORD_ENCJPEG_RGB_FB_ADDR_REG(jpeg->base_addr));
        printf("   > jpeg frame buffer:%x.\n", IORD_ENCJPEG_JPEG_FB_ADDR_REG(jpeg->base_addr));
/*****************************************************************************
 * Ethernet Controller Initialization
 *****************************************************************************/
```

```
  printf(" - Initializing uIP...\n");
  memcpy(ip_config.uip_mac_addr.addr, "\x00\x1f\x33\xa4\x8d\x89", 6);
  uip_ipaddr(ip_config.uip_ip_addr, 192,168,1,200);
  uip_ipaddr(ip_config.uip_ip_gw, 192,168,1,1);
  uip_ipaddr(ip_config.uip_ip_mask, 255,255,255,0);
  if(net_init(&ip_config) != 0)          goto app_exit;
  /*******************************************************************************
   * Application
   *******************************************************************************/
  printf(" - Application started.\n");
  while(1)
  {
              net_schedule();
              monitor_schedule();
  }
  app_exit:
  printf("Application exit.\n");
  exit(1);
  return 0;
}
```

Lista de Código 21 - Ficheiro main.c

A função global ***_log*** tem como finalidade dar suporte ao processo de *debug* de toda a solução pois na implementação do *device driver* esta está definida como externa. No meu entender, é uma boa estratégia uma vez que aqui podemos, em *running time*, redirecionar o *output* das mensagens geradas bem como ativar ou desativar. Uma alternativa, mais eficiente, será redirecionar este *output* pata um *logic device* e então efetuar *routing* das mensagens para um, ou mais, *physic devices*. Desta forma, desabilitar o envio de mensagens seria efetuar *routing* para o *null device* e ao ativar poderíamos enviar as mensagens para um ficheiro ou até, para um destino remoto.

A inicialização da interface *Sockets* é feita recorrendo à função ***net_init***, que recebe no argumento os parâmetros *IPv4* do sistema para que o núcleo **uIP** possa ser configurado através da função ***uip_setup***. A implementação do servidor *Web* também é inicializada, através da função ***monitor_init*** que inicializa alguns dos atributos do *array* ***monitor_stack*** e das respetivas *threads*.
Depois de inicializados ambos os componentes, a função ***main***, fica a executar um *loop* que consiste em consecutivas evocações das funções ***net_schedule*** e ***monitor_schedule***, responsáveis por todo o processamento de rede e sessões *http* respetivamente.

Análise da área ocupada na *FPGA*

Após síntese do sistema podemos verificar a área ocupada na *FPGA*, que pode ser obtida diretamente do sumário de síntese da ferramenta *QuartosII*.

Área Ocupada na FPGA	
Elementos Lógicos	13,280 (12%)
Registos	8748
Pins	169 (32%)
Bits de Memória	110,358 (3%)
PLL´s	1 (25%)
Multiplicadores de 9 bits	2 (1%)

Figura 28 - Àrea ocupada na FPGA

Podemos verificar que a FPGA aqui utilizada (Altera Cyclone® IV 4CE115 FPGA device), após síntese do sistema, ainda ficou com a maior parte de área diasponível devendo-se em grande parte ao facto de a memória do sistema ser externa a este dispositivo.

Dispositivos de memória do sistema

Apesar de não ter sido focado o dispositivo *SRAM*, o sistema dispõe de um dispositivo deste tipo, onde reside o *frame buffer RGB*, atualizado pelo módulo *BAYER*.

Os dispositivos de memória do sistema podem ser visualizados acedendo ao *url* http://192.168.1.200/memory.html de onde podemos verificar que o sistema dispõe de 8MBytes em memória *SDRAM* e 2MBytes de memória *SRAM*.

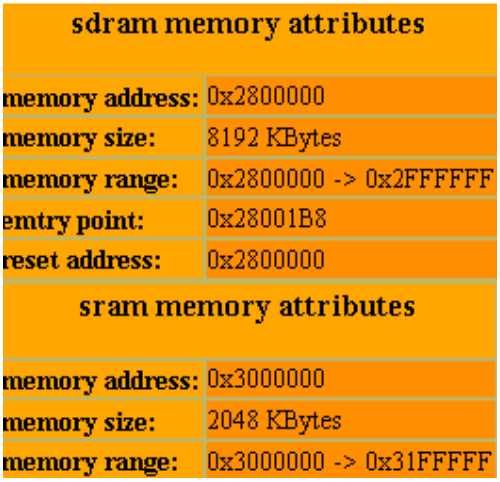

Figura 29 - Interface http - Memória do Sistema

Análise da Configuração do *linker*

O *linker* é responsável pela organização, em memória, das diversas secções do ficheiro executável, bem como de diversos *labels* entre os quais o *entry point* que corresponde ao endereço de arranque do sistema.

Quando o sistema arranca o controlo do processador é passado para o troço de código apontado pelo *entry point* **start** que está definido no ficheiro **crt0.S** que, depois de efetuar as inicializações de baixo nível do sistema (vulgarmente chamado o módulo de arranque), chama a função **alt_main()** cuja implementação está no ficheiro *alt_main.c*.
A função **alt_main()**, que tem por objetivo efetuar as inicializações do sistema a um nível mais alto (io, etc), antes de chamar a função **main()** para passar o controlo do processador ao programador, recorre à implementação que consta no ficheiro **alt_sys_init.c** para chamar as funções **alt_irq_init()**, que inicializam o controlador de interrupções através do seu *device driver*, e **alt_sys_init()**, que inicializa o *timer* definido na *macro ALT_SYS_CLK* (caso exista) recorrendo ao seu *device driver*.
A página *html* de memória do sistema também apresenta a configuração do *linker* onde é determinado o mapeamento, em memória, das diversas secções do programa, de onde podemos verificar que estas secções estão todas mapeadas na *SDRAM*.

system memory map

section	range	size
.exceptions	0x2800020 -> 0x28001B8	102
.text	0x28001B8 -> 0x2867B78	106096
.rodata	0x2867BE8 -> 0x286ED68	7264
.rwdata	0x286ED68 -> 0x2872E20	4142
.bss	0x2876ED8 -> 0x2884F94	14383
.stack/heap	0x2884F94 -> 0x2FFF000	1914 KBytes

Figura 30 - Interface http - secções do programa

Componentes do sistema
Os componentes do sistema podem ser visualizados acedendo ao *url* http://192.168.1.200/system.html.

Nesta página podemos verificar a listagem dos componentes instanciados na *FPGA*. Estes componentes, do ponto de vista do processador *NIOS*, são periféricos acedidos pelos respetivo *device drivers* através dos seus endereços.

System Page

system attributes

device	name	address
sys id controller	/dev/sysid	0x1000040
jtag uart controller	/dev/jtag_uart	0x1000048
pll controller	/dev/pll	0x3201500
std in	/dev/jtag_uart	0x1000048
std out	/dev/jtag_uart	0x1000048
std error	/dev/jtag_uart	0x1000048
timer controller	/dev/timer	0x1000000
bayer pattern controller	/dev/bayer_pattern_0	0x3201480
jpeg encoder	/dev/encjpeg_0	0x32014c0
i2c controller	/dev/i2c	0x1000020
ethernet controller	/dev/tse_0	0x3201000
sg-dma rx controller	/dev/sgdma_rx	0x3201440
sg-sgdma tx controller	/dev/sgdma_tx	0x3201400
sdram controller	/dev/sdram	0x2800000
sram controller	/dev/sram	0x3000000

cpu attributes

architecture:	altera_nios2
id:	0
clock[MHz]:	83
exception address:	0x2800020
reset address:	0x2800000

Figura 31 – Interface http – sistema

Performance em rede

Para verificar a performance em rede verificamos, durante uma sessão de *Streaming* na interface *http* do sistema, qual o débito conseguido através da ferramenta *Wireshark*. Na figura que se segue, podemos verificar que o sistema consegue atingir picos na ordem dos 250 Kbps durante uma sessão de *Streaming*.

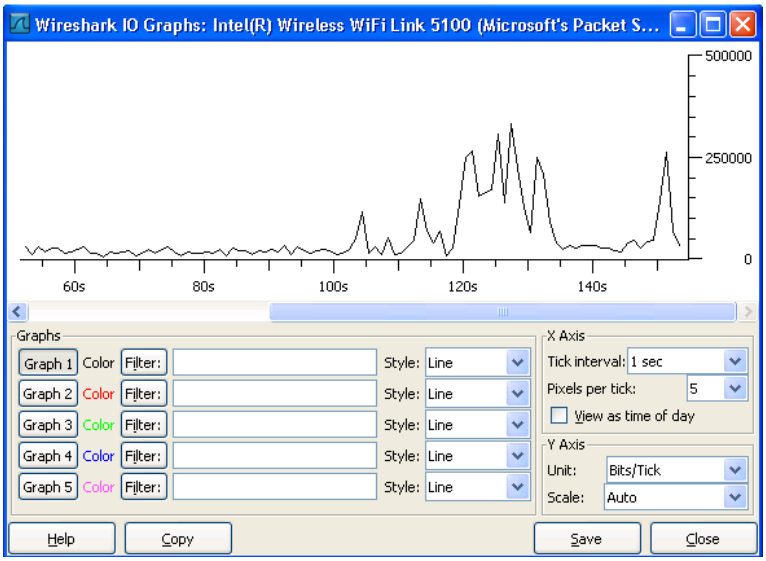

Figura 32 - Performance em Rede – Wireshark

Visualização de Imagem

Conforme referido no capítulo 6, a visualização de imagem é feita através de uma aplicação cliente que suporte o formato *MJPEG*, como por exemplo o *Firefox*.

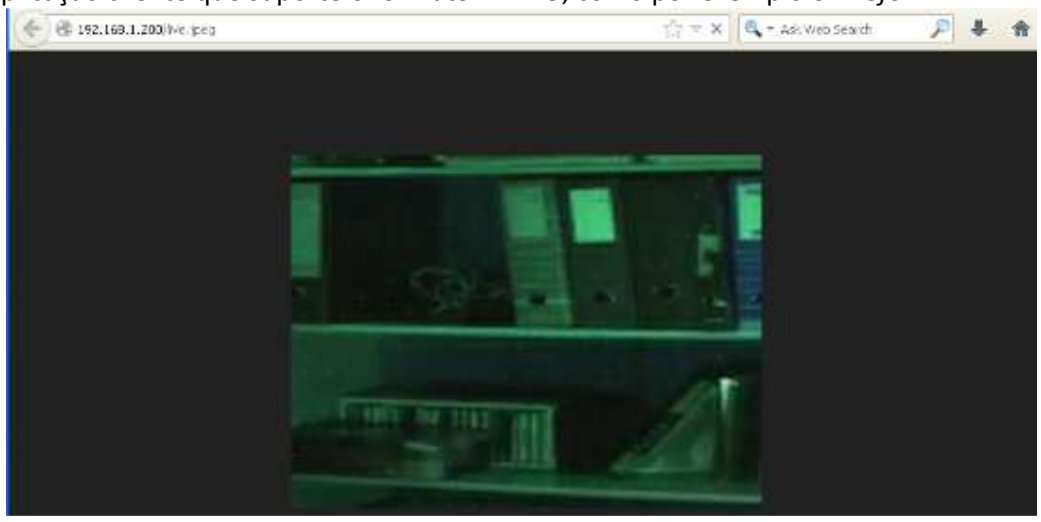

Figura 33 - Visualização de Imagem - Firefox

Capítulo 7 – Conclusões

Ao longo deste trabalho foram sendo adicionados componentes ao sistema para que este cumprisse os pressupostos iniciais e tornou- se cada vez mais evidente a importância das *FPGA* neste tipo de projetos uma vez que se verifica de que é perfeitamente possível implementar um sistema, com complexidade relevante, num único circuito integrado. Pois note-se que a Câmara de Rede aqui desenvolvida é constituída apenas pela *FPGA*, dispositivos de memória externa a pelo módulo *Ethernet* e este nível de integração nunca teria sido conseguido se recorrêssemos a um microcontrolador.

O sistema sintetizado dispõe do módulo *ENCJPEG* cuja intenção era implementar a compressão das imagens captadas pela Câmara. No entanto, apesar de a aplicação servidora estar implementada por forma a, também suportar este dispositivo, não foi possível coloca- lo a funcionar.
O recurso à biblioteca *libjpeg*, que implementa as mesmas funções, foi a alternativa encontrada mas o sistema, devido às suas limitações de processamento, oferece um ritmo de tramas muito abaixo do desejável.

Durante o desenvolvimento deste Trabalho de Project foram sendo tomadas decisões acerca do modo como os módulos deveriam de funcionar com o intuito de se obter o sistema com as características a que nos propusemos no início.
Após conclusão da implementação existe sempre espaço para melhorias tanto em termos de desempenho do sistema como em termos de conceção.
No que diz respeito ao desempenho, as interfaces adotadas para transferência de dados (*Avalon-MM Master*) entre os módulos (*D5M* e *EV_JPEG_ENC*) e o sistema, poderiam ter sido escolhidas por forma a obter transferências com taxas mais elevadas.
Em termos de conceção, em vez de desenvolver o *stack TCP/IP* e interface *socket* poderíamos ter optado por efetuar *porting* de uma distribuição de *Linux*, como por exemplo o *uCLinux*, e recorrer à implementação nativa da interface *socket*.
Do ponto de vista académico, a opção aqui adotada é bastante mais desafiante mas, do ponto de vista comercial, a opção pelo *porting* teria sido melhor.

Referências

[1] Nios II Processor Reference Handbook

[2] Getting Started with Nios II Software in Eclipse

[3] Embedded Peripherals IP User Guide.

[4] DE2_115_User_manual

[5] Exception Handling.

[6] SOPC Builder User Guide.

[7] Getting Started with Nios II Software in Eclipse.

[8] HAL API Reference.

[9] Nios II Hardware Development Tutorial.

[10] Nios II Processor Reference Handbook.

[11] Nios II Software Developer Handbook.

[12] Embedded Design Handbook.

[13] Developing Programs Using the Hardware Abstraction Layer.

[14] Embedded Peripherals IP User Guide.

[15] Using the SDRAM Memory on Altera's DE2-115 Board with VHDL Design

[16] Avalon Interface Specifications

[17] Sitio Online: http://pt.wikipedia.org/wiki/TCP/IPhttp://pt.wikipedia.org/wiki/TCP/IP

[18] Sitio Online: http://en.wikipedia.org/wiki/UIP_%28micro_IP%29

[19] Making SOPC Builder Components

[20] Triple-Speed Ethernet MegaCore Function User Guide

[21] AN10216-01 I2C MANUAL

[22] TRDB_D5M User Guide

[23] Sitio Online: http://en.wikipedia.org/wiki/Address_Resolution_Protocol

[24] EV_JPEG_ENC IP Core Specification

[25] Sitio Online: http://en.wikipedia.org/wiki/Motion_JPEGhttp://en.wikipedia.org/wiki/Motion_JPEG

[26] Sitio Online:

http://en.wikipedia.org/wiki/Libjpeghttp://en.wikipedia.org/wiki/Libjpeg

[27] Sitio Online: http://www.axis.com/en/products

[28] Nios II C2H Compiler User Guide

[29] Triple-Speed Ethernet MegaCore Function User Guide

Anexo 1 – Aplicação para atendimento de interrupções

```
#include <sys/alt_irq.h>
#include <system.h>
#include <altera_avalon_pio_regs.h>
#include <alt_types.h>

struct leds
{
	char state;
	char dummy[3];
};
volatile struct leds leds_state;

#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
	static void toogle_leds(void * context)
#else
	static void toogle_leds(void* context, alt_u32 id)
#endif
{
	//Cast context to struct leds *
	volatile struct leds * leds_state_ptr = (volatile struct leds *) context;

	if(leds_state_ptr->state == 0)
	{
		IOWR_ALTERA_AVALON_PIO_SET_BITS(PIO_LED_BASE, 0x1);
		leds_state_ptr->state = 1;
	}
	else
	{
		IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(PIO_LED_BASE, 0x1);
		leds_state_ptr->state = 0;
	}
	// Reset the SW0 capture register.
	IOWR_ALTERA_AVALON_PIO_EDGE_CAP(PIO_SWITCH_BASE, 0x1);
}

int main()
{
	//Initialize green led and state flag
	IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(PIO_LED_BASE, 0x1);
	leds_state.state = 0;

	// Enable SW0 interrupt.
	IOWR_ALTERA_AVALON_PIO_IRQ_MASK(PIO_SWITCH_BASE, 0x1);
	// Reset the SW0 edge capture register.
	IOWR_ALTERA_AVALON_PIO_EDGE_CAP(PIO_SWITCH_BASE, 0x1);
	// Register the interrupt handler.
	#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
		alt_ic_isr_register(PIO_SWITCH_IRQ_INTERRUPT_CONTROLLER_ID,PIO_SWITCH_IRQ, toogle_leds,
(void*)&leds_state,0x0);
	#else
		alt_irq_register( PIO_SWITCH_IRQ, (void*)&leds_state,		toogle_leds);
	#endif

	while(1){};
	return 0;
}
```

Anexo 1 - Função main para atendimento de interrupções

Anexo 2 – Aplicação para Relógio de sistema

```
#include <sys/alt_irq.h>
#include <system.h>
#include <altera_avalon_pio_regs.h>
#include <altera_avalon_timer_regs.h>
#include <alt_types.h>
#include <sys/alt_alarm.h>

struct leds
{
	char state;
	char dummy[3];
};
volatile struct leds leds_state;
static alt_alarm alarm;
/*
*			The callback function.
* The return value of the registered callback
* function is the number of ticks until the next call to callback
*/
static alt_u32 toogle_leds(void * context)
{
	//Cast context to struct leds *
	volatile struct leds * leds_state_ptr = (volatile struct leds *) context;

	if(leds_state_ptr->state == 0)
	{
		IOWR_ALTERA_AVALON_PIO_SET_BITS(PIO_LED_BASE, 0x1);
		leds_state_ptr->state = 1;
	}
	else
	{
		IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(PIO_LED_BASE, 0x1);
		leds_state_ptr->state = 0;
	}
	/* This function is called once per second */
	return alt_ticks_per_second()*TIMER_0_MULT;
}

int main()
{
	//Initialize green led and state flag
	IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(PIO_LED_BASE, 0x1);
	leds_state.state = 0;

	// Register the alarm.
	if (alt_alarm_start (&alarm, alt_ticks_per_second()*TIMER_0_MULT, \
				toogle_leds, (void *)&leds_state) < 0)
	{
		return -1;
	}

	while(1){};
	return 0;
}
```

Anexo 2 - Aplicação para Relógio de sistema

Anexo 3 – Device Driver do Módulo Ethernet - marvell_88e111.c

```
#include <sys/alt_cache.h>
#include <sys/alt_irq.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <altera_avalon_sgdma.h>
#include <altera_avalon_sgdma_regs.h>
```

```
#include "marvell_88e1111.h"

/*
 * ethernet global variables
 */
extern void _log(const char * format, va_list arg);
extern alt_tse_mac_group *pmac_groups[TSE_MAX_MAC_IN_SYSTEM];
alt_tse_system_info tse_mac_device[MAXNETS];
ALT_LLIST_HEAD(marvell_88E1111_dev_list);
/*
 * internal ethernet api implementation
 */
void eth_log(eth_dev * dev, alt_u32 debug_level, const char * fmt, ...)
{
        if(dev != NULL)
                if(dev->debug_level & debug_level)
                {
                        va_list _va_list;
                        va_start(_va_list, fmt);
                        _log(fmt, _va_list);
                        va_end(_va_list);
                }
}
alt_dev* eth_open_dev(const char* name, int iface)
{
 alt_dev* dev = (alt_dev*)alt_find_dev(name, &marvell_88E1111_dev_list);
 if(dev != NULL)
 {
          ((eth_dev*)dev)->iface = iface;
 }
 return dev;
}
void eth_rx_sgdma_isr(void *context)
{
        alt_u32 sgdma_status, sgdma_control;
        //cast for current device
        volatile eth_dev * dev = (eth_dev*) context;
        if(dev == NULL) return;

        //current descriptor chain with sanity check.
        volatile eth_desc_chain * pchain = &dev->rx_queue.chain_list[dev->rx_queue.list_loop];
        //get sg-dma status flag
        sgdma_status        = IORD_ALTERA_AVALON_SGDMA_STATUS(dev->mi.rx_sgdma->base);
        sgdma_control       = IORD_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.rx_sgdma->base);
        //clear sg-dma status flag
        IOWR_ALTERA_AVALON_SGDMA_STATUS(dev->mi.rx_sgdma->base, 0x17);

        eth_log((eth_dev *)dev, ETH_DEBUG_ISR,"---------------------------------------------\n");
        eth_log((eth_dev *)dev, ETH_DEBUG_ISR, "isr entry-->sg-dma status:0X%X sg-dma control:0X%X chain(%u, %u)\n", \
                        sgdma_status, sgdma_control, \
                        dev->rx_queue.list_loop, pchain->chain_loop);

        /*
         * the RX SGDMA device have errors
         */
        if(sgdma_status & ALTERA_AVALON_SGDMA_STATUS_ERROR_MSK)
        {
                //handle errors on sgdma device
                eth_log((eth_dev *)dev, ETH_DEBUG_ISR, "isr entry-->sg-dma with errors!\n");

                //clear sg-dma RUN flag
                IOWR_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.rx_sgdma->base, \
                        (IORD_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.rx_sgdma->base) & \
                         ~ALTERA_AVALON_SGDMA_CONTROL_RUN_MSK) );

                // wait untill sg-dma controller is not busy
                while ( (IORD_ALTERA_AVALON_SGDMA_STATUS(dev->mi.rx_sgdma->base) & \
                        ALTERA_AVALON_SGDMA_STATUS_BUSY_MSK) ){}

                //resume copy from rx FIFO into rx sgdma descriptor chain
                tse_mac_aRxRead( (tse_mac_trans_info *)&dev->mi, (alt_sgdma_descriptor *)&pchain->desc[pchain-
```

```
>chain_loop]);

                        //clear STOP_DMA_ER flag
                        IOWR_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.rx_sgdma->base, \
                                                IORD_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.rx_sgdma->base) & \
                                                (~ALTERA_AVALON_SGDMA_CONTROL_STOP_DMA_ER_MSK));


                        ++dev->mib.rx_sgdma_errors;
                }
                /*
                 * Handle received packet(s)
                 */
                /*else*/ if(sgdma_status & (ALTERA_AVALON_SGDMA_STATUS_DESC_COMPLETED_MSK | \
                                        ALTERA_AVALON_SGDMA_STATUS_EOP_ENCOUNTERED_MSK))
                {
                        eth_log((eth_dev *)dev, ETH_DEBUG_ISR, "isr entry-->Descriptor Status:0x%X\n", \
                                IORD_ALTERA_TSE_SGDMA_DESC_STATUS(&pchain->desc[pchain->chain_loop]) & 0xff);

                        if(IORD_ALTERA_TSE_SGDMA_DESC_STATUS(&pchain->desc[pchain->chain_loop]) & \
                                        ALTERA_AVALON_SGDMA_DESCRIPTOR_STATUS_TERMINATED_BY_EOP_MSK)
                        {
                                ++dev->mib.rx_ok;
                        }
                        else//descriptor with errors
                        {
                                //drop the packet. set the flag OWNED_BY_HW so the packet will not be send to tcp/ip stack
                                IOWR_ALTERA_TSE_SGDMA_DESC_CONTROL( \
                                                (alt_sgdma_descriptor*)&pchain->desc[pchain->chain_loop], \
                                                IORD_ALTERA_TSE_SGDMA_DESC_CONTROL(&pchain->desc[pchain-
>chain_loop]) | \
                                                ALTERA_AVALON_SGDMA_DESCRIPTOR_CONTROL_OWNED_BY_HW_MSK);
                                //increment mib rx_errors
                                ++dev->mib.rx_errors;
                        }
                        //increment chain loop
                        ++pchain->chain_loop;
                }
                /*
                 * the chain is full so we must swap to another chain in the list
                 */
                if(sgdma_status & ALTERA_AVALON_SGDMA_STATUS_CHAIN_COMPLETED_MSK)
                {
                        eth_log((eth_dev *)dev, ETH_DEBUG_ISR, "isr entry-->chain list %d completed!\n", dev->rx_queue.list_loop);
                        //check if we are at the end of the chain list
                        if(++dev->rx_queue.list_loop == ETH_RX_CHAIN_LIST_SIZE)
                        {
                                //initialize chain list loop
                                dev->rx_queue.list_loop = 0;
                        }
                        //actualize pointer for the next chain
                        pchain = &dev->rx_queue.chain_list[dev->rx_queue.list_loop];
                        //set OWNED_BY_HW flag on descriptor chain
                        alt_u32 chain_loop;
                        alt_8 rx_desc_control;
                        alt_u32 rx_discard = dev->mib.rx_discard;
                        for(chain_loop = 0; chain_loop < ETH_RX_DESC_CHAIN_SIZE; chain_loop++)
                        {
                                //get current control of the descriptor
                                rx_desc_control =IORD_ALTERA_TSE_SGDMA_DESC_CONTROL(&pchain->desc[chain_loop]);
                                //verify if the packet was send to the tcp/ip stack. If so OWNED_BY_HW flag is all ready set
                                if((rx_desc_control & ALTERA_AVALON_SGDMA_DESCRIPTOR_CONTROL_OWNED_BY_HW_MSK)
== 0x0)
                                {
                                        ++dev->mib.rx_discard;
                                }
                                IOWR_ALTERA_TSE_SGDMA_DESC_CONTROL( \
                                        &pchain->desc[chain_loop], \
                                        ALTERA_AVALON_SGDMA_DESCRIPTOR_CONTROL_GENERATE_EOP_MSK | \
                                        ALTERA_AVALON_SGDMA_DESCRIPTOR_CONTROL_OWNED_BY_HW_MSK);
                        }
```

```
                        //last chain descriptor have OWNED_BY_HW flag cleared
                        IOWR_ALTERA_TSE_SGDMA_DESC_CONTROL( &pchain->desc[chain_loop], 0x0);
                        pchain->chain_loop = 0;
                        tse_mac_aRxRead( (tse_mac_trans_info *)&dev->mi, \
                                        (alt_sgdma_descriptor *)&pchain->desc[pchain->chain_loop]);

                        eth_log((eth_dev *)dev, ETH_DEBUG_ISR, "isr entry-->%d frames discarted!\n", dev->mib.rx_discard -
rx_discard);
                }
}
alt_u32 * eth_desc_mem(eth_dev * dev, alt_u32 chain_list, alt_u32 desc)
{
                //determine memory used for each descriptor
                alt_u32 mem_per_desc = ETH_RX_DESC_MEM_OFFSET;
                if(dev->queue_mem_base == HEAP)
                {
                        int mem_base = (int)alt_uncached_malloc(mem_per_desc + 0x1f);
                        if(mem_base == NULL)            return NULL;
                        //mem_base += (mem_base%0x20);
                        mem_base = (mem_base + 0x1f) & ~0x1f;
                        return (alt_u32 *)mem_base;
                }
                else
                {
                        //determine memory used for each chain
                        alt_u32 mem_per_chain = ETH_RX_DESC_CHAIN_SIZE * mem_per_desc;
                        //determine relative address of the list
                        alt_u32 mem_list_rel_addr = chain_list * mem_per_chain;
                        //determine relative address of the specified descriptor in the list
                        alt_u32 mem_desc_rel_addr = (desc * mem_per_desc) + mem_list_rel_addr;
                        //if(mem_desc_rel_addr > dev->queue_mem_base + dev->queue_mem_span)
                        if(mem_desc_rel_addr > dev->queue_mem_span)
                                return NULL;
                        return (alt_u32 *)alt_remap_cached ( (volatile void*) (dev->queue_mem_base + mem_desc_rel_addr), 4
/*mem_per_desc*/);
                }
}
alt_32 eth_sgdma_init(eth_dev * dev)
{
                volatile eth_desc_chain * pchain;
                alt_u32 *uncached_packet_payload;
                /*
                 * Stop RX SGDMA device
                 * this issue is very important for stability reasons!!!
                 */
                eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "perform rx sgdma device [%p] stop...", \
                                        dev->mi.rx_sgdma->base);
                // Make sure RX SGDMA controller is not busy
                while ( (IORD_ALTERA_AVALON_SGDMA_STATUS(dev->mi.rx_sgdma->base) & \
                        ALTERA_AVALON_SGDMA_STATUS_BUSY_MSK) ){}
                //clear RUN flag
                IOWR_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.rx_sgdma->base, \
                                (IORD_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.rx_sgdma->base) & \
                                 ~ALTERA_AVALON_SGDMA_CONTROL_RUN_MSK) );
                //Clear any (previous) status register information
                IOWR_ALTERA_AVALON_SGDMA_STATUS(dev->mi.rx_sgdma->base, 0xFF);
                eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "done!\n");
                /*
                 * Stop TX SGDMA device
                 * this issue is very important for stability reasons!!!
                 */
                eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "perform tx sgdma device [%p] stop...", \
                                        dev->mi.tx_sgdma->base);
                // Make sure TX SGDMA controller is not busy
                while ( (IORD_ALTERA_AVALON_SGDMA_STATUS(dev->mi.tx_sgdma->base) & \
                        ALTERA_AVALON_SGDMA_STATUS_BUSY_MSK) ){}
                //clear RUN flag
                IOWR_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.tx_sgdma->base, \
                                (IORD_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.tx_sgdma->base) & \
                                 ~ALTERA_AVALON_SGDMA_CONTROL_RUN_MSK) );
                //Clear any (previous) status register information
```

```
	IOWR_ALTERA_AVALON_SGDMA_STATUS(dev->mi.tx_sgdma->base, 0xFF);
	eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "done!\n");
	/*
	 * rx sgdma descriptor chain initialization
	 */
	eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "perform rx sgdma descriptor chain initialization...");
	alt_u32 list_loop, chain_loop;
	//loop on descriptor list
	for(list_loop = 0; list_loop < ETH_RX_CHAIN_LIST_SIZE; list_loop++)
	{
		pchain = &dev->rx_queue.chain_list[list_loop];
		//loop on descriptor chain
		for(chain_loop = 0; chain_loop < ETH_RX_DESC_CHAIN_SIZE; chain_loop++)
		{
			// ensure bit-31 of tse_ptr->pkt_array[tse_ptr->chain_loop]->nb_buff is clear before passing
			// to SGDMA Driver
			uncached_packet_payload = eth_desc_mem(dev, list_loop, chain_loop);
			if(uncached_packet_payload == NULL)
			{
				eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "error!\n");
				eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "could not allocate memory for
descriptor(%u, %u)\n", \
							(unsigned)list_loop, (unsigned)chain_loop);
				return ETHERNET_ERROR;
			}
			alt_avalon_sgdma_construct_stream_to_mem_desc( \
					(alt_sgdma_descriptor *)&pchain->desc[chain_loop], \
					(alt_sgdma_descriptor *)&pchain->desc[chain_loop + 1], \
				uncached_packet_payload,	//	write addr
				0,		// read until EOP
				0);		// don't write to constant address

			IOWR_ALTERA_TSE_SGDMA_DESC_CONTROL( \
					&pchain->desc[chain_loop], \
					ALTERA_AVALON_SGDMA_DESCRIPTOR_CONTROL_GENERATE_EOP_MSK |
\
					ALTERA_AVALON_SGDMA_DESCRIPTOR_CONTROL_OWNED_BY_HW_MSK);
		}//descriptor chain loop
		//initialize the chain_loop of the current list_loop
		pchain->chain_loop = 0;
		//last chain descriptor have OWNED_BY_HW flag cleared
		IOWR_ALTERA_TSE_SGDMA_DESC_CONTROL( &pchain->desc[chain_loop], 0x0);
	}//descriptor list loop
	//initialize descriptor list loop
	dev->rx_queue.list_loop = 0;
	//initialize packet_mux
	dev->rx_queue.packet_mux.chain_loop = 0;
	dev->rx_queue.packet_mux.list_loop = 0;
	dev->rx_queue.packet_mux.pchain = (eth_desc_chain *)&dev->rx_queue.chain_list[0];
	eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "done!\n");
	/*
	 * tx sgdma descriptor initialization
	 * only checks if we have memory to the tx descriptor and initialize second descriptor
	 */
	eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "perform tx sgdma descriptor initialization...");
	uncached_packet_payload = eth_desc_mem(dev, ETH_RX_CHAIN_LIST_SIZE, 0);
	if(uncached_packet_payload == NULL)
	{
		eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "error!\n");
		eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "could not allocate memory for tx descriptor!!!\n");
		return ETHERNET_ERROR;
	}
	//last chain descriptor have OWNED_BY_HW flag cleared
	IOWR_ALTERA_TSE_SGDMA_DESC_CONTROL( &dev->tx_queue.desc[1], 0x0);
	eth_log((eth_dev *)dev, ETH_DEBUG_SGDMA, "done!\n");
	return ETHERNET_SUCCESS;
}
/*
 * ethernet api implementation
 */
alt_32 eth_close(alt_fd* fd)
```

```
{
	eth_dev * dev = (eth_dev *) fd;
	//Uninstall SGDMA (RX) interrupt handler.
	eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "perform rx sgdma device interrupt handler uninstall...");
	alt_avalon_sgdma_register_callback(dev->mi.rx_sgdma, NULL, 0, NULL);
	eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "done!\n");
	//Disable Receive path on the device
	IOWR_ALTERA_TSEMAC_CMD_CONFIG(dev->mi.base, \
				IORD_ALTERA_TSEMAC_CMD_CONFIG(dev->mi.base) & \
				~ALTERA_TSEMAC_CMD_RX_ENA_MSK );

	return ETHERNET_SUCCESS;
}
alt_32 eth_sgdma_open(eth_dev * dev)
{
	volatile eth_desc_chain * pchain = &dev->rx_queue.chain_list[dev->rx_queue.list_loop];
	alt_32 rx_sgdma_control, tx_sgdma_control, result;
	/*
	 * config RX SGDMA device and install interrupt handler.
	 */
	//enable polling mode so the SGDMA device will poll until the first descriptor with OWNED_BY_HW is set
	//when an OWNED_BY_HW cleared is find SGDMA device handles an IE_CHAIN_COMPLETED
	rx_sgdma_control =   ALTERA_AVALON_SGDMA_CONTROL_IE_ERROR_MSK | \
					ALTERA_AVALON_SGDMA_CONTROL_IE_GLOBAL_MSK | \

	ALTERA_AVALON_SGDMA_CONTROL_IE_CHAIN_COMPLETED_MSK | \

	ALTERA_AVALON_SGDMA_CONTROL_IE_DESC_COMPLETED_MSK | \

	ALTERA_AVALON_SGDMA_CONTROL_IE_EOP_ENCOUNTERED_MSK;
	IOWR_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.rx_sgdma->base, rx_sgdma_control);

	eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "perform rx sgdma device interrupt handler installation...");
	alt_avalon_sgdma_register_callback(dev->mi.rx_sgdma, \
				(alt_avalon_sgdma_callback)eth_rx_sgdma_isr, 0,
				(void*)dev);
	eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "done!\n");
	/*
	 * config TX SGDMA device
	 */
	tx_sgdma_control =   0x0;
	IOWR_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.tx_sgdma->base, tx_sgdma_control);

	/*
	 * output tx and rx sgdma devices control and status registers
	 */
	alt_u32 rx_sgdma_dev_control = IORD_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.rx_sgdma->base);
	eth_print_reg32((eth_dev *)dev, rx_sgdma_dev_control, "RX SGDMA DEVICE CONTROL REGISTER:");
	alt_u32 rx_sgdma_dev_status = IORD_ALTERA_AVALON_SGDMA_STATUS(dev->mi.rx_sgdma->base);
	eth_print_reg32((eth_dev *)dev, rx_sgdma_dev_status, "RX SGDMA DEVICE STATUS REGISTER:");
	alt_u32 tx_sgdma_dev_control = IORD_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.tx_sgdma->base);
	eth_print_reg32((eth_dev *)dev, tx_sgdma_dev_control, "TX SGDMA DEVICE CONTROL REGISTER:");
	alt_u32 tx_sgdma_dev_status = IORD_ALTERA_AVALON_SGDMA_STATUS(dev->mi.tx_sgdma->base);
	eth_print_reg32((eth_dev *)dev, tx_sgdma_dev_status, "TX SGDMA DEVICE STATUS REGISTER:");
	/*
	 * initialize mib status for the iface
	 */
	dev->mib.tx_bytes = 0;
	dev->mib.rx_bytes = 0;
	dev->mib.tx_errors = 0;
	dev->mib.rx_errors = 0;
	dev->mib.rx_ok = 0;
	dev->mib.tx_discard = 0;
	dev->mib.rx_discard = 0;
	dev->mib.rx_sgdma_errors = 0;
	/*
	 * starting Asynchronous copy from rx FIFO into rx sgdma descriptor chain
	 */
	eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "Perform Asynchronous copy from rx FIFO into rx sgdma descriptor
chain(%u, %u)...\n", \
				(unsigned)dev->rx_queue.list_loop, (unsigned)pchain->chain_loop);
```

```
	if(tse_mac_aRxRead( (tse_mac_trans_info *)&dev->mi, (alt_sgdma_descriptor *)&pchain->desc[pchain->chain_loop])
== SUCCESS)
	{
		//eth_log("done!\n");
		result = ETHERNET_SUCCESS;
	}
	else
	{
		eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "error!\n");
		result = ETHERNET_ERROR;
	}
	//clear STOP_DMA_ER flag
	IOWR_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.rx_sgdma->base, \
			IORD_ALTERA_AVALON_SGDMA_CONTROL(dev->mi.rx_sgdma->base) & \
			(~ALTERA_AVALON_SGDMA_CONTROL_STOP_DMA_ER_MSK));
	return result;
}
alt_32 eth_config(eth_dev * dev, alt_u8 * _eth_addr)
{
	alt_u32 cmd_config, result;
	/*
	 * check speed and reset the PHY if necessary
	 */
	eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "Perform physical reset... ");
	result = getPHYSpeed((np_tse_mac *)dev->mi.base);
	if((result & 0x00ff00) != 0)
	{
		//error found so we will print error...
		if((result & ALT_TSE_E_NO_PMAC_FOUND) != 0)
			eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "Argument *pmac not found from the list of MAC
detected during init\n");
		if((result & ALT_TSE_E_NO_MDIO) != 0)
			eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "No MDIO used by the MAC\n");
		if((result & ALT_TSE_E_NO_PHY) != 0)
			eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "No PHY detected\n");
		if((result & ALT_TSE_E_NO_COMMON_SPEED) != 0)
			eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "No common speed found for Multi-port MAC\n");
		if((result & ALT_TSE_E_AN_NOT_COMPLETE) != 0)
			eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "PHY auto-negotiation not completed\n");
		if((result & ALT_TSE_E_NO_PHY_PROFILE) != 0)
			eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "No PHY profile match the detected PHY\n");
		if((result & ALT_TSE_E_PROFILE_INCORRECT_DEFINED) != 0)
			eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "PHY Profile not defined correctly\n");
		if((result & ALT_TSE_E_INVALID_SPEED) != 0)
			eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "Invalid speed read from PHY\n");
		return ETHERNET_ERROR;
	}
	eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "done!\n");
	/*
	 * reset the mac
	 */
	eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "Perform mac reset... ");
	IOWR_ALTERA_TSEMAC_CMD_CONFIG(dev->mi.base, ALTERA_TSEMAC_CMD_CNT_RESET_MSK);
	//whait for sw reset bit clear
	alt_32 timer = 0;
	while(IORD_ALTERA_TSEMAC_CMD_CONFIG(dev->mi.base) & \
	      ALTERA_TSEMAC_CMD_SW_RESET_MSK)
	{
		++ timer;
		if(timer > ALTERA_CHECKLINK_TIMEOUT_THRESHOLD)
			break;
	}
	if(timer > ALTERA_CHECKLINK_TIMEOUT_THRESHOLD)
	{
		eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "\nsw reset clear timeout!!!\n");
		return ETHERNET_ERROR;
	}
	//read mac command_config
	cmd_config = IORD_ALTERA_TSEMAC_CMD_CONFIG(dev->mi.base);
	if((cmd_config & 0x3) != 0)
	{
```

```
                eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "\nRX and TX not disabled after reset!!!\n");
                eth_print_reg32((eth_dev*)dev, cmd_config, "TSEMAC_CMD_CONFIG REGITER:");
                return ETHERNET_ERROR;
        }
        eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "done!\n");
        /*
         * config physical
         */
        eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "Perform physical configuration... ");
        //physical speed configuration
        switch((result & 0xf) >> 1)
        {
                case 0x4:  eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "10Mbps ");
                                        //cmd_config |= ALTERA_TSEMAC_CMD_ETH_SPEED_MSK;
                                        cmd_config = ALTERA_TSEMAC_CMD_ENA_10_MSK;
                                        break;
                case 0x2:  eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "100Mbps ");
                                        //cmd_config |= ALTERA_TSEMAC_CMD_ETH_SPEED_MSK;
                                        //cmd_config |= ALTERA_TSEMAC_CMD_ENA_10_MSK;
                                        break;
                case 0x1:  eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "1000Mbps ");
                                        cmd_config = ALTERA_TSEMAC_CMD_ETH_SPEED_MSK;
                                        //cmd_config |= ALTERA_TSEMAC_CMD_ENA_10_MSK;
                                        break;
                default:   eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "unknow bit rate ");
        };
        //physical duplex configuration
        switch(result & 0x1)
        {
                case 0x0:  eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "Half Duplex...");
                                        cmd_config |= ALTERA_TSEMAC_CMD_HD_ENA_MSK;
                                        break;
                default:   eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "Full Duplex...");
        };
        IOWR_ALTERA_TSEMAC_CMD_CONFIG(dev->mi.base, cmd_config);
        eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "done!\n");
        /*
         * Start SG-DMA devices - from now they process data from, and to, tse fifos
         */
        result = eth_sgdma_open(dev);
        /*
         * config mac addr registers
         */
        eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "Perform mac address configuration... ");
        memcpy(dev->dev_addr, _eth_addr, sizeof(dev->dev_addr));
        alt_32 mac = 0;
        /* upper word (bits 47:16) of MAC address */
        mac = ((alt_32)dev->dev_addr[0]) & 0xff;
        mac |= ((alt_32)dev->dev_addr[1])<< 8;
        mac |= ((alt_32)dev->dev_addr[2])<< 16;
        mac |= ((alt_32)dev->dev_addr[3])<< 24;
        IOWR_ALTERA_TSEMAC_MAC_0(dev->mi.base,mac);
        mac = IORD_ALTERA_TSEMAC_MAC_0(dev->mi.base);
        eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "0x:%.2X-%.2X-%.2X-%.2X-",(unsigned)mac & 0xff, \
                        (unsigned)(mac >> 8) & 0xff, (unsigned)(mac >> 16) & 0xff, (unsigned)(mac >> 24) & 0xff);
        /* lower half-word (bits 15:0) of MAC address.*/
        mac = ((alt_32)dev->dev_addr[4]) & 0xff;
        mac |= ((alt_32)dev->dev_addr[5])<< 8;
        IOWR_ALTERA_TSEMAC_MAC_1(dev->mi.base,mac & 0xffff);
        mac = IORD_ALTERA_TSEMAC_MAC_1(dev->mi.base);
        eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "%.2X-%.2X...",(unsigned)mac & 0xff, (unsigned)(mac >> 8) & 0xff);
        eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "done!\n");
        /*
         * config fifo configuration registers
         */
        //mac registers for fifo according manual TSE manual Pag 75
        eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "Perform fifo configuration... ");
IOWR_ALTERA_TSEMAC_FRM_LENGTH(dev->mi.base, ALTERA_TSE_MAC_MAX_FRAME_LENGTH);

IOWR_ALTERA_TSEMAC_TX_SECTION_EMPTY(dev->mi.base, \
                TSE_0_TRANSMIT_FIFO_DEPTH - 16); //1024/4;
```

```
    IOWR_ALTERA_TSEMAC_TX_ALMOST_FULL(dev->mi.base,  3);
    IOWR_ALTERA_TSEMAC_TX_ALMOST_EMPTY(dev->mi.base, 8);
    //Cut Throught Mode, Set this Threshold to 0 to enable Store and Forward Mode
    IOWR_ALTERA_TSEMAC_TX_SECTION_FULL(dev->mi.base,  0);

    IOWR_ALTERA_TSEMAC_RX_SECTION_EMPTY(dev->mi.base, \
                        TSE_0_RECEIVE_FIFO_DEPTH - 16);
    IOWR_ALTERA_TSEMAC_RX_ALMOST_FULL(dev->mi.base, 8);
    IOWR_ALTERA_TSEMAC_RX_ALMOST_EMPTY(dev->mi.base, 8);
    //Cut Throught Mode, Set this Threshold to 0 to enable Store and Forward Mode
    IOWR_ALTERA_TSEMAC_RX_SECTION_FULL(dev->mi.base,  /*16*/\
                        TSE_0_RECEIVE_FIFO_DEPTH*4/ALTERA_TSE_MAC_MAX_FRAME_LENGTH);
            eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "done!\n");
            /*
             * TSEMAC_TX_CMD_STAT register
             * tcp/ip stack will provide CRC on transmited frames
             */
            if((dev->cfg & ETH_OMIT_CRC) != 0)
                        IOWR_ALTERA_TSEMAC_TX_CMD_STAT(dev->mi.base, \
                                    (IORD_ALTERA_TSEMAC_TX_CMD_STAT(dev->mi.base)) | \
                                    ALTERA_TSEMAC_TX_CMD_STAT_OMITCRC_MSK);
            else
                        IOWR_ALTERA_TSEMAC_TX_CMD_STAT(dev->mi.base, \
                                    (IORD_ALTERA_TSEMAC_TX_CMD_STAT(dev->mi.base)) & \
                                    (~ALTERA_TSEMAC_TX_CMD_STAT_OMITCRC_MSK));
            /*
             * config command_config register and start TX and RX operation
             */
            eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "Perform command_config register configuration... ");
            //some flags of command_config register was all ready configurated on
            //eth_config routine. So we must save them.
            cmd_config = IORD_ALTERA_TSEMAC_CMD_CONFIG(dev->mi.base);
            //active Promiscuous mode
            if((dev->cfg & ETH_PROMIS_ENABLE) != 0)
                        cmd_config |= ALTERA_TSEMAC_CMD_PROMIS_EN_MSK;
            //loop enable
            if((dev->cfg & ETH_LOOPBACK_ENABLE) != 0)
                        cmd_config |= ALTERA_TSEMAC_CMD_LOOPBACK_MSK;
            //Set this bit to 1 to omit length checking
            if((dev->cfg & ETH_NO_LENGTH_CHECK) != 0)
                        cmd_config |= ALTERA_TSEMAC_CMD_NO_LENGTH_CHECK_MSK;
            //Do not remove padding from receive frames
            //cmd_config &= ~ALTERA_TSEMAC_CMD_PAD_EN_MSK;
            //forwarding crc on receive frames to the user
            if((dev->cfg & ETH_CRC_FWD) != 0)
                        cmd_config |= ALTERA_TSEMAC_CMD_CRC_FWD_MSK;
            //Set this bit to 0 to retain the source MAC address in transmit frames
            cmd_config &= ~ALTERA_TSEMAC_CMD_TX_ADDR_INS_MSK;
            //start tx and rx operation
            cmd_config |= (ALTERA_TSEMAC_CMD_TX_ENA_MSK | ALTERA_TSEMAC_CMD_RX_ENA_MSK);
            //write command config
            IOWR_ALTERA_TSEMAC_CMD_CONFIG(dev->mi.base, cmd_config);
            eth_log((eth_dev *)dev, ETH_DEBUG_CONFIG, "done!\n");
            //eth_print_reg32((eth_dev*)dev, cmd_config, "TSEMAC_CMD_CONFIG REGITER:");
            eth_print_reg32((eth_dev*)dev, \
                                    IORD_ALTERA_TSEMAC_CMD_CONFIG(dev->mi.base), \
                                    "TSEMAC_CMD_CONFIG REGITER:");
            return result;
}
int eth_init(eth_dev * dev, int queue_location, int queue_size, alt_u8 debug)
{
            int iface = dev->iface;
            dev->debug_level = debug;
            dev->queue_mem_base = queue_location;
            dev->queue_mem_span = queue_size;
            /*
             * initialize config flags and debug level default values
             */
            eth_log((eth_dev *)dev, ETH_DEBUG_SYSTEM, "perform device driver default initialization...");
            //cfg flag´s default values. user must change it before eth_config routine call
            dev->cfg = ETH_NO_LENGTH_CHECK | ETH_CRC_FWD;
```

```
	eth_log((eth_dev *)dev, ETH_DEBUG_SYSTEM, "done!\n");
	/*
	 * adding iface to the system
	 */
	eth_log((eth_dev *)dev, ETH_DEBUG_SYSTEM, \
				"perform iface[%u] addition to the system...", iface);

	alt_tse_system_mac iface_system_mac = {TSE_SYSTEM_MAC(TSE_0)};
	alt_tse_system_sgdma iface_system_sgdma = {TSE_SYSTEM_SGDMA(SGDMA_TX, SGDMA_RX)};
	alt_tse_system_desc_mem iface_system_desc_mem = {TSE_SYSTEM_NO_DESC_MEM()};
	alt_tse_system_shared_fifo iface_system_shared_fifo = {TSE_SYSTEM_NO_SHARED_FIFO()};
	alt_tse_system_phy iface_system_phy = {TSE_SYSTEM_PHY(ETH0_MDIO, NULL)};
	if(alt_tse_system_add_sys(&iface_system_mac, &iface_system_sgdma, &iface_system_desc_mem, \
				&iface_system_shared_fifo, &iface_system_phy) != SUCCESS)
	{
		eth_log((eth_dev *)dev, ETH_DEBUG_SYSTEM, "alt_tse_system_add_sys error...\n");
		return ETHERNET_ERROR;
	}

	eth_log((eth_dev *)dev, ETH_DEBUG_SYSTEM, "done!\n");
	//configure MDIO with RGMII protocol
	marvell_cfg_rgmii((np_tse_mac *)tse_mac_device[iface].tse_mac_base);
	/*
	 * SGDMA RX and TX device
	 */
	eth_log((eth_dev *)dev, ETH_DEBUG_SYSTEM, "perform tx and rx sgdma devices initialization...");
	alt_sgdma_dev * rx_sgdma_dev;
	alt_sgdma_dev * tx_sgdma_dev;
	rx_sgdma_dev = alt_avalon_sgdma_open(tse_mac_device[iface].tse_sgdma_rx);
	tx_sgdma_dev = alt_avalon_sgdma_open(tse_mac_device[iface].tse_sgdma_tx);
	if(rx_sgdma_dev == NULL || tx_sgdma_dev == NULL)
	{
		eth_log((eth_dev *)dev, ETH_DEBUG_SYSTEM, "alt_avalon_sgdma_open error...\n");
		return ETHERNET_ERROR;
	}
	//rx and tx sgdma_dev initialization. mi points to rx_sgdma and tx_sgdma
	tse_mac_initTransInfo2(		/*&dev->mi*/&dev->mi, \
							tse_mac_device[iface].tse_mac_base, \
							(alt_32)tx_sgdma_dev, \
							(alt_32)rx_sgdma_dev, \
							0);
	eth_log((eth_dev *)dev, ETH_DEBUG_SYSTEM, "done! rx:%p, tx:%p\n", \
				dev->mi.rx_sgdma->base, dev->mi.tx_sgdma->base);
	//initialize rx descriptors
	if(eth_sgdma_init(dev) == ETHERNET_SUCCESS)
	{
		return ETHERNET_SUCCESS;
	}
	else
	{
		return ETHERNET_ERROR;
	}
	return ETHERNET_ERROR;
}
int eth_rcv(alt_fd* fd, char * data, int len)
{
	eth_dev * dev = (eth_dev *) fd;

	int pk_size = 0;
	//current chain in RGDMA device...do not touch!!!
	volatile eth_desc_chain * pchain		= &dev->rx_queue.chain_list[dev->rx_queue.list_loop];
	//ethernet mux for this iface
	volatile eth_packet_mux * pmux			= &dev->rx_queue.packet_mux;
	/*
	 * check if we need to move mux to next chain
	 */
	if(pmux->chain_loop == ETH_RX_DESC_CHAIN_SIZE)
	{
		//move mux to next chain
		pmux->chain_loop = 0;
		++pmux->list_loop;
```

```
                        if(pmux->list_loop == ETH_RX_CHAIN_LIST_SIZE){pmux->list_loop = 0;}
                        pmux->pchain = (eth_desc_chain *)&dev->rx_queue.chain_list[pmux->list_loop];
                }
                /*
                 * Handle the packet
                 */
                alt_u8 desc_control =IORD_ALTERA_TSE_SGDMA_DESC_CONTROL((alt_sgdma_descriptor *)&pmux->pchain-
>desc[pmux->chain_loop]);
                //verify if the mux is pointing to current chain in RGDMA device and to the same descriptor
                if((pmux->pchain == pchain) && (pmux->chain_loop == pchain->chain_loop))
                {
                        //do not touch
                        //eth_log((eth_dev *)dev, ETH_DEBUG_RX, "no frames to handle!\n");
                        pk_size = 0;
                        return pk_size;
                }
                else
                {
                        //we can touch on the descriptor...
                        eth_log((eth_dev *)dev, ETH_DEBUG_RX, \
                                        "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n");
                        eth_log((eth_dev *)dev, ETH_DEBUG_RX, "checking chain(%u, %u) for new frames...", \
                                        (unsigned)pmux->list_loop, \
                                        (unsigned)pmux->chain_loop);
                        if((desc_control & ALTERA_AVALON_SGDMA_DESCRIPTOR_CONTROL_OWNED_BY_HW_MSK))
                        {
                                //The descriptor have errors
                                eth_log((eth_dev *)dev, ETH_DEBUG_RX, "current descriptor have errors!\n");
                                pk_size = 0;
                        }
                        else
                        {
                                //The descriptor have no errors.
                                //handle and set the OWNED_BY_HW flag
                                pk_size = pmux->pchain->desc[pmux->chain_loop].actual_bytes_transferred;
                                eth_log((eth_dev *)dev, ETH_DEBUG_RX, "frame with %u bytes sended to tcp/ip stack!\n", \
                                                (unsigned)pmux->pchain->desc[pmux-
>chain_loop].actual_bytes_transferred);
                                if(pk_size > len)
                                {
                                        //packet data is greater than buffer size
                                        memcpy(data, (char *)pmux->pchain->desc[pmux->chain_loop].write_addr, len);
                                }
                                else
                                {
                                        //packet data is less than buffer size
                                        memcpy(data, (char *)pmux->pchain->desc[pmux->chain_loop].write_addr, pk_size);
                                }
                                //actualize mib rx_bytes
                                //dev->mib.rx_bytes += pk_size;
                                //set the OWNED_BY_HW flag to acknowledge on isr. so mib rx_discard will not count
                                IOWR_ALTERA_TSE_SGDMA_DESC_CONTROL(&pmux->pchain->desc[pmux->chain_loop], \
                                                ALTERA_AVALON_SGDMA_DESCRIPTOR_CONTROL_OWNED_BY_HW_MSK);

                        }
                        eth_log((eth_dev *)dev, ETH_DEBUG_RX, "frame statistic: %u ok; %u with error.\n", \
                                                (unsigned)IORD_ALTERA_TSEMAC_A_FRAMES_RX_OK(dev->mi.base), \
                                                (unsigned)IORD_ALTERA_TSEMAC_IF_IN_ERRORS(dev->mi.base));
                        eth_log((eth_dev *)dev, ETH_DEBUG_RX, "sgdma status: 0x%x; descriptor control: 0x%x; descriptor status:
0x%x\n", \
                                                (unsigned)IORD_ALTERA_AVALON_SGDMA_STATUS(dev->mi.rx_sgdma-
>base), \
                                                (unsigned)IORD_ALTERA_TSE_SGDMA_DESC_CONTROL(&pmux->pchain-
>desc[pmux->chain_loop]), \
                                                (unsigned)IORD_ALTERA_TSE_SGDMA_DESC_STATUS(&pmux->pchain-
>desc[pmux->chain_loop]));
                        //jump to the next descriptor
                        ++pmux->chain_loop;
                }
                return pk_size;
}
```

```
int eth_send(alt_fd* fd, const char* data, int len)
{
	eth_dev * dev = (eth_dev *) fd;
	alt_u32 *uncached_packet_payload;
	alt_u32 tx_length, desc_status, desc_control_1, desc_control_2, sgdma_status;

	eth_log((eth_dev *)dev, ETH_DEBUG_TX, \
				">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n");
	eth_log((eth_dev *)dev, ETH_DEBUG_TX, "frame with %u bytes received from tcp/ip stack...", \
				(unsigned)len);
	//check buffer len
	if(len > (ETH_MAX_FRAME_LEN))
	{
		eth_log((eth_dev *)dev, ETH_DEBUG_TX, "too big for tx descriptor!!!\n");
		return 0;
	}
	//check if TX SGDMA is busy
	if(IORD_ALTERA_AVALON_SGDMA_STATUS(dev->mi.tx_sgdma->base) & ALTERA_AVALON_SGDMA_STATUS_BUSY_MSK)
	{
		eth_log((eth_dev *)dev, ETH_DEBUG_TX, "TX SGDMA device busy\n!!!\n");
		++dev->mib.tx_discard;
		return 0;
	}
	//get, and check, free space from memory (sgdma_desc_mem- system internal memory)
	uncached_packet_payload = eth_desc_mem(dev, ETH_RX_CHAIN_LIST_SIZE, 0);
	if(uncached_packet_payload == NULL)
	{
		eth_log((eth_dev *)dev, ETH_DEBUG_TX, "could not allocate memory for tx descriptor!!!\n");
		return 0;
	}
	//copy data to the memory and contruct the descriptor
	memcpy(uncached_packet_payload, data, len);
	uncached_packet_payload = (alt_u32 *)alt_remap_cached ( (volatile void*)uncached_packet_payload, 4/*len*/);
	alt_avalon_sgdma_construct_mem_to_stream_desc( \
				(alt_sgdma_descriptor *)&dev->tx_queue.desc[0], \
				(alt_sgdma_descriptor *)&dev->tx_queue.desc[1], \
				uncached_packet_payload, \
				len,
	//bytes to send
	    0,				// don't read from constant address
	    1,				// generate sop (start of packet)
	    1,				// generate endofpacket signal
	    0);				// atlantic channel (don't know/don't care: set to 0)
	//send data to tx fifo
	desc_control_1 = IORD_ALTERA_TSE_SGDMA_DESC_CONTROL(&dev->tx_queue.desc[0]);
	tx_length = tse_mac_sTxWrite((tse_mac_trans_info *)&dev->mi, (alt_sgdma_descriptor *)&dev->tx_queue.desc[0]);
	desc_control_2 = IORD_ALTERA_TSE_SGDMA_DESC_CONTROL(&dev->tx_queue.desc[0]);
	desc_status = IORD_ALTERA_TSE_SGDMA_DESC_STATUS(&dev->tx_queue.desc[0]);
	sgdma_status = IORD_ALTERA_AVALON_SGDMA_STATUS(dev->mi.tx_sgdma->base);
	/*
	 * clear tx_sgdma device status register
	 */
	IOWR_ALTERA_AVALON_SGDMA_STATUS(dev->mi.tx_sgdma->base, \
				ALTERA_AVALON_SGDMA_STATUS_ERROR_MSK | \
				ALTERA_AVALON_SGDMA_STATUS_DESC_COMPLETED_MSK | \
				ALTERA_AVALON_SGDMA_STATUS_CHAIN_COMPLETED_MSK | \
				ALTERA_AVALON_SGDMA_STATUS_BUSY_MSK);

	if(tx_length == ENP_RESOURCE)
	{
		//verify erros
		eth_log((eth_dev *)dev, ETH_DEBUG_TX, "descarted!!!\n");
		++dev->mib.tx_discard;
		return 0;
	}
	else
	{
		eth_log((eth_dev *)dev, ETH_DEBUG_TX, "copy %u bytes from dec. memory into transmit FIFO\n", \
					(unsigned)tx_length);
		dev->mib.tx_bytes += tx_length;
	}
```

```
	eth_log((eth_dev *)dev, ETH_DEBUG_TX, "frame statistic: %u ok; %u with error.\n", \
					(unsigned)IORD_ALTERA_TSEMAC_A_FRAMES_TX_OK(dev->mi.base), \
					(unsigned)IORD_ALTERA_TSEMAC_IF_OUT_ERRORS(dev->mi.base), \
					IORD_ALTERA_TSE_SGDMA_DESC_STATUS(&dev->tx_queue.desc[0]));
	eth_log((eth_dev *)dev, ETH_DEBUG_TX, "sgdma status: 0x%x; descriptor control: 0x%x/%x; descriptor status: 0x%x\n",
\
					(unsigned)sgdma_status, (unsigned)desc_control_1, (unsigned)desc_control_2,
(unsigned)desc_status);
	//printf("-------------------------------------------------\n");
	return (int)tx_length;
}
```

Anexo 3 - Device Driver do Módulo Ethernet - marvell_88e111.c

Anexo 4 - Aplicação Arp Request - icmp.c

```
#include <system.h>
#include <alt_types.h>
#include <stdarg.h>
#include <stdio.h>
#include "marvell_88e1111.h"
#include "sniffer.h"
#include <sys/alt_alarm.h>
#include <string.h>

void _log(const char * format, va_list arg)
{
	vprintf(format, arg);
}

volatile eth_driver * pinst;

alt_u8 frame[ETH_MAX_FRAME_LEN];
alt_u8 arp_request[14 + 28];

alt_u8 sender_mac[] = {0x00, 0x1f, 0x33, 0xa4, 0x8d, 0x89};
alt_u8 sender_ipv4[] = {192, 168, 0, 200};
alt_u8 target_ipv4[] = {192, 168, 0, 10};
static alt_alarm alarm;
alt_u16 semaphore;
static alt_u32 send_arp(void * context)
{
	volatile eth_driver * _pinst = (eth_driver *)context;
	_pinst->raw_send(arp_request, sizeof(arp_request), ETH0);
	/* This function is called once per second */
	return (10 * alt_ticks_per_second());
}

void count_received_packets()
{
	++semaphore;
}

void create_arp_request()
{
	alt_u16 i = 0;
	memcpy(&arp_request[i], "\xff\xff\xff\xff\xff\xff", 6);  //DA
	i += 6;
	memcpy(&arp_request[i], sender_mac, 6);                 //SA
	i += 6;
	memcpy(&arp_request[i], "\x08\x06", 2);                 //Type
	i += 2;
	memcpy(&arp_request[i], "\x00\x01", 2);                 //HTYPE
	i += 2;
	memcpy(&arp_request[i], "\x08\x00", 2);                 //PTYPE
	i += 2;
	memcpy(&arp_request[i], "\x06", 1);                     //HLEN
	i += 1;
	memcpy(&arp_request[i], "\x04", 1);                     //PLEN
	i += 1;
```

```
		memcpy(&arp_request[i], "\x00\x01", 2);			//OPER - replay
		i += 2;
		memcpy(&arp_request[i], sender_mac, 6);							//Sender hardware
address (SHA)
		i += 6;
		memcpy(&arp_request[i], sender_ipv4, 4);		//Sender protocol address (SPA)
		i += 4;
		memcpy(&arp_request[i], "\x00\x00\x00\x00\x00\x00", 6);	//Target hardware address (THA)
		i += 6;
		memcpy(&arp_request[i], target_ipv4, 4);		//Target protocol address (TPA)
}
int main()
{
		semaphore = 0;
		/*
		 * initialize ethernet device
		 */
		if((pinst = eth_init(ETH0, count_received_packets)) == NULL)
		{
			printf("eth_init error!!!\n");
			return -1;
		}
		pinst->debug_level = ETH_DEBUG_TX | ETH_DEBUG_RX | ETH_DEBUG_CONFIG | ETH_DEBUG_SYSTEM;
		if(pinst->config(sender_mac, ETH0) == ETHERNET_ERROR)
		{
			printf("eth_config error!!!\n");
			return -1;
		}
		printf("eth device ok\n");
		create_arp_request();
		/*
		 * Register the alarm to sen an arp request every second
		 */
		if (alt_alarm_start (&alarm, 10 * alt_ticks_per_second(), send_arp, (void *)pinst) < 0)
		{
			printf("alarm registering error!!!\n");
			return -1;
		}
		/*
		 * Sniffing received packets
		 */
		alt_u16 size = 0;
		while(1)
		{
			while(semaphore == 0){}//this should be an passive waiting!!!
			size = pinst->pkt_rcv(frame, sizeof(frame), ETH0);
			if(size != 0)
			{
				sniffer(frame, size);
			}
			--semaphore;
		};
		return 0;
}
```

Anexo 4 - Aplicação Arp Request - icmp.c

Anexo 5 – Interface Socket - net.c

```
#define NDEBUG
#include <assert.h>
#include <string.h>
#include <stdarg.h>
#include "net.h"

extern int uip_setup(uip_ip_config * ip);
extern void uip_activity();
/* The uip_conns array holds all TCP connections. */
```

```
extern struct uip_conn uip_conns[UIP_CONF_MAX_CONNECTIONS];
extern u16_t uip_listenports[UIP_CONF_MAX_LISTENPORTS];
extern void _log(const char * format, va_list arg);
socket sockets[UIP_CONF_MAX_CONNECTIONS];

/*
 * log support
 */
void net_logf(const char *m, ...)
{
	/*
	 * O uIP invoca a função uip_log quando precisa de emitir mensagens.
	 */
	va_list _va_list;
	va_start(_va_list, m);
	_log(m, _va_list);
	va_end(_va_list);
}
/*############################################################################################
 *                                                                    SOCKET PRIVATE API
 *############################################################################################*/

static void net_window_init(socket *s)
{
	int i;
	for(i = 0; i < SOCKET_WINDOW_SIZE; i++)
	{
		memset(&s->window[i].ack_seq[0], 0, 4);
		s->window[i].len = 0;
	}
}
static int net_window_full(socket *s)
{
	int i;
	for(i = 0; i < SOCKET_WINDOW_SIZE; i++)
	{
		if(s->window[i].len == 0)	break;
	}
	return (i == SOCKET_WINDOW_SIZE - 1)? 1: -1;
}
static u16_t net_window_add(socket *s)
{
	u16_t len = 0;
	int i;
	for(i = 0; i < SOCKET_WINDOW_SIZE; i++)
	{
		if(s->window[i].len == 0)	break;
		len += s->window[i].len;
	}
	assert(i != SOCKET_WINDOW_SIZE - 1);
	memcpy(&s->window[i].ack_seq[0], &s->conn->snd_nxt[0], 4);
	s->window[i].len = s->sent_size;
	len += s->window[i].len;
	return len;
}
static u16_t net_window_ack(socket *s)
{
	u16_t len = 0;
	int i, j;
	for(i = 0; i < SOCKET_WINDOW_SIZE; i++)
	{
		len += s->window[i].len;
		if(s->window[i].ack_seq[0] == s->conn->snd_nxt[0] && s->window[i].ack_seq[1] == s->conn->snd_nxt[1]
&& \
					s->window[i].ack_seq[2] == s->conn->snd_nxt[2] && s->window[i].ack_seq[3] == s-
>conn->snd_nxt[3])	break;
	}
	//assert(i != SOCKET_WINDOW_SIZE);

	for(j = 0; j < SOCKET_WINDOW_SIZE; j++)
	{
```

```
                        ++i;
                        if(i < SOCKET_WINDOW_SIZE)
                        {
                                memcpy(&s->window[j].ack_seq[0], &s->window[i].ack_seq[0], 4);
                                s->window[j].len = s->window[i].len;
                        }
                        else
                        {
                                memset(&s->window[j].ack_seq[0], 0, 4);
                                s->window[j].len = 0;
                        }
                }
                return len;
}
static u16_t net_window_rexmit(socket *s)
{
                u16_t len = 0;
                int i;
                for(i = 0; i < SOCKET_WINDOW_SIZE; i++)
                {
                        if(s->window[i].ack_seq[0] == s->conn->rcv_nxt[0] && s->window[i].ack_seq[1] == s->conn->rcv_nxt[1] &&
\
                                                s->window[i].ack_seq[2] == s->conn->rcv_nxt[2] && s->window[i].ack_seq[3] == s-
>conn->rcv_nxt[3])    break;
                        len += s->window[i].len;
                }
                net_window_init(s);
                //assert(i != SOCKET_WINDOW_SIZE);

                return len;
}
static u16_t net_window_sent(socket *s)
{
                u16_t len = 0;
                int i;
                for(i = 0; i < SOCKET_WINDOW_SIZE; i++)
                {
                        if(s->window[i].len == 0)               break;
                        len += s->window[i].len;
                }
                //assert(i != SOCKET_WINDOW_SIZE);

                return len;
}
static void net_handle_output(socket *s)
{
                assert(s->conn == uip_conn);
                u16_t window_sent = 0;
                /*
                 * connection just been connected
                 */
                if(uip_connected() > 0)
                {
                        net_window_init(s);
                        s->sent_size = (uip_initialmss()/*uip_mss()*/ > fbuffer_count((Fbuffer *)&s->output_fbuffer))? \
                                        fbuffer_count((Fbuffer *)&s->output_fbuffer) : /*uip_mss();*/uip_initialmss();
                        s->sent_ptr = (u8_t*)fbuffer_read_block_ptr((Fbuffer *)&s->output_fbuffer);
                }
                /*
                 * connection already connected
                 */
                else
                {
                        //insert last segment into transmission window
                        if(s->sent_size > 0)
                        {
                                net_window_add(s);
                        }
                        if(uip_acked() > 0)
                        {
                                fbuffer_read_seek((Fbuffer *)&s->output_fbuffer, net_window_ack(s));
```

```
                }
                else if(uip_rexmit() > 0)
                {
                        fbuffer_read_seek((Fbuffer *)&s->output_fbuffer, net_window_rexmit(s));
                }
                window_sent = net_window_sent(s);

                if(fbuffer_count((Fbuffer *)&s->output_fbuffer) > window_sent)
                {
                        s->sent_size = (uip_mss() > fbuffer_count((Fbuffer *)&s->output_fbuffer) - window_sent)? \
                                fbuffer_count((Fbuffer *)&s->output_fbuffer) - window_sent : uip_mss();
                }
                else
                {
                        s->sent_size = 0;
                }
                s->sent_ptr = (u8_t*)(fbuffer_read_block_ptr((Fbuffer *)&s->output_fbuffer) + window_sent);
        }
        if(net_window_full(s) < 0 && s->sent_size > 0)
        {
                uip_send(s->sent_ptr, s->sent_size);

                net_logf("[net_handle_output (%p)]->buffer[%u/%u] %u bytes sent. %u bytes left.\n", \
                        s, \
                        fbuffer_count((Fbuffer *)&s->output_fbuffer),\
                        SOCKET_BUFFER_SIZE,\
                        s->sent_size, \
                        ((u16_t)(fbuffer_count((Fbuffer *)&s->output_fbuffer)& 0xffff)) - (s->sent_size + window_sent));
        }
        /*
        else
        {
                s->sent_size = 0;
        }
        */
}
/*
 * copy received data from uip uip_appdata to socket input_buffer
 */
static void net_handle_input(socket *s)
{
        assert(s->conn == uip_conn);
        if(uip_newdata())
        {
                if(fbuffer_write_block((Fbuffer *)&s->input_fbuffer,(char*)uip_appdata, uip_datalen()) < uip_datalen())
                {
                        net_logf("[net_handle_input (%p)]->incoming buffer overflow[%u/%u] %u bytes loss.\n", \
                                        s, \
                                        fbuffer_count((Fbuffer *)&s->input_fbuffer), \
                                        UIP_CONF_BUFFER_SIZE, \
                                        uip_datalen() - fbuffer_free((Fbuffer *)&s->input_fbuffer));
                }
        }
}
/*
 * Perform uip handler
 */
void net_handle_connection(void)
{
        /*
         * Check if is a new connection
         */
        if(uip_connected())
        {
                net_logf("[net_handle_connection]->new connection from:%d.%d.%d.%d:%d->%d ... ", \
                                uip_ipaddr1(uip_conn->ripaddr), uip_ipaddr2(uip_conn->ripaddr), \
                                uip_ipaddr3(uip_conn->ripaddr), uip_ipaddr4(uip_conn->ripaddr), \
                                HTONS(uip_conn->rport), \
                                HTONS(uip_conn->lport));
                //find a free socket
                int i;
```

```
                for(i = 0; i < UIP_CONF_MAX_CONNECTIONS; i++)
                {
                        if(sockets[i].state == SOCKET_STATE_FREE_MASK)
                        {
                                break;
                        }
                }
                if(i == UIP_CONF_MAX_CONNECTIONS)
                {//did not find free socket
                        net_logf("did not find a free socket!!!\n");
                        uip_abort();
                        return;
                }
                else
                {//find free socket
                        uip_conn->appstate = (void *)&sockets[i];
                        sockets[i].conn = uip_conn;
                        sockets[i].sent_size = 0;
                        sockets[i].sent_ptr = (u8_t*)&sockets[i].output_buffer[0];
                        //initialize frame buffers
                        fbuffer_init((Fbuffer *)&sockets[i].input_fbuffer, (char *)&sockets[i].input_buffer[0], \
                                        sizeof(sockets[i].input_buffer) - 1);
                        fbuffer_init((Fbuffer *)&sockets[i].output_fbuffer, (char *)&sockets[i].output_buffer[0], \
                                        sizeof(sockets[i].output_buffer) - 1);
                        net_logf("socket[%u (%p)] allocated.\n", i, &sockets[i]);
                        sockets[i].state = SOCKET_STATE_ACCEPT_MASK;
                }
        }
        volatile socket * s = (socket *)uip_conn->appstate;
        if(s->conn == uip_conn)
        {
                if(uip_closed() > 0 || uip_aborted() > 0 || uip_timedout() > 0)
                {
                        net_logf("[net_handle_connection (%p)]->connection %s.\n", ((socket *)uip_conn->appstate), \
                                        (uip_closed())? "closed" : (uip_aborted())? "aborted" : "timeout");
                        //release socket
                        ((socket *)uip_conn->appstate)->state = SOCKET_STATE_FREE_MASK;
                        ((socket *)uip_conn->appstate)->conn = NULL;
                        uip_conn->appstate = NULL;
                        return;
                }
                if(s->state == SOCKET_STATE_CLOSE_MASK)
                {
                        if(fbuffer_count((Fbuffer *)&s->output_fbuffer) == 0)
                        {
                                uip_close();
                                //uip_abort();
                                return;
                        }
                }
                net_handle_input(s);
                net_handle_output(s);
        }
        else
        {
                if(uip_closed() > 0 || uip_aborted() > 0 || uip_timedout() > 0)
                {
                        //Unreferenced socket. No problem ...
                }
                else
                {
                        net_logf("[net_handle_connection]->connection from:%d.%d.%d.%d:%d->%d ...we should not
be there!!!\n", \
                                uip_ipaddr1(uip_conn->ripaddr), uip_ipaddr2(uip_conn->ripaddr), \
                                uip_ipaddr3(uip_conn->ripaddr), uip_ipaddr4(uip_conn->ripaddr), \
                                HTONS(uip_conn->rport), \
                                HTONS(uip_conn->lport));
                        uip_abort();
                }
                return;
        }
```

```
	return;
}
/*############################################################################################
 *                                                                      SOCKET PUBLIC API
 *############################################################################################*/
/*
 *
 */
void net_schedule(void)
{
	uip_activity();
}
/*
 * Set socket state to SOCKET_STATE_CLOSE_MASK
 * The responsability to close the connection is net_handle_connection
 */
void net_close(socket * s)
{
	assert(s->conn != NULL);
	if(s->conn != NULL)
	{
		assert(s->state == SOCKET_STATE_CONNECTED_MASK);
		if(s->state == SOCKET_STATE_CONNECTED_MASK)
		{
			s->state = SOCKET_STATE_CLOSE_MASK;
		}
		else
		{
			net_logf("[net_close (%p)]->socket is not connected.\n", s);
		}
	}
	else
	{
		net_logf("[net_close]->we should not be there!!!\n");
	}
}
int net_init(net_ip_config * ip_config)
{
	/*
	 * uip configuration
	 */
	int result = uip_setup(ip_config);
	/*
	 * uip_conns[UIP_CONNS] and monitor_state[UIP_CONNS] initialization
	 */
	int i;
	for(i = 0; i < UIP_CONF_MAX_CONNECTIONS; i++)
	{
		sockets[i].conn = NULL;
		uip_conns[i].appstate = NULL;
		sockets[i].state = SOCKET_STATE_FREE_MASK;
	}
	return result;
}
/*
 * Scan sockets for SOCKET_STATE_ACCEPT_MASK sockets and return it
 */
socket * net_server_accept(int port)
{
	int i;
	/*
	 * check if port is all ready on uip listen ports
	 */
	/*
	for(i = 0; i < UIP_CONF_MAX_LISTENPORTS; i++)
	{
		if(uip_listenports[i] == HTONS(port))
			break;
	}
	if(i ==UIP_CONF_MAX_LISTENPORTS)
		uip_listen(HTONS(port));
```

```
	*/
	uip_listen(HTONS(port));
	/*
	 * now we try to find a socket with state SOCKET_STATE_ACCEPT_MASK
	 * and, if local port is equal to port, return the socket
	 */
	for(i = 0; i < UIP_CONF_MAX_CONNECTIONS; i++)
	{
		if(sockets[i].state == SOCKET_STATE_ACCEPT_MASK)
		{
			assert(sockets[i].conn != NULL);
			if(	sockets[i].conn->lport == HTONS(port) && \
					sockets[i].conn != NULL)
			{
				net_logf("[net_server_accept (%p)]->connection  accepted.\n", &sockets[i]);
				sockets[i].state = SOCKET_STATE_CONNECTED_MASK;
				return &sockets[i];
			}
		}
	}
	return NULL;
}
s32_t net_recv(socket * s, char * buffer, u16_t length)
{
	assert(s->conn != NULL);
	if(s->conn != NULL)
	{
		if(s->state == SOCKET_STATE_CONNECTED_MASK)
		{
			return fbuffer_read_block((Fbuffer *)&s->input_fbuffer, buffer, length);
		}
		else
		{
			net_logf("[net_recv (%p)]->socket is not connected.\n", s);
			return INVALID_SOCKET;
		}
	}
	else
	{
		net_logf("[net_recv (%p)]->we should not be there...please check if socket is connected!\n", s);
		return INVALID_SOCKET;
	}
	return INVALID_SOCKET;
}
s32_t net_send(socket * s, char * buffer, u16_t length)
{
	assert(s->conn != NULL);
	if(s->conn != NULL)
	{
		if(s->state == SOCKET_STATE_CONNECTED_MASK)
		{
			return fbuffer_write_block((Fbuffer *)&s->output_fbuffer, buffer, length);
		}
		else
		{
			net_logf("[net_send (%p)]->socket is not connected.\n", s);
			return INVALID_SOCKET;
		}
	}
	else
	{
		net_logf("[net_send (%p)]->we should not be there...please check if socket is connected!\n", s);
		return INVALID_SOCKET;
	}
	return INVALID_SOCKET;
}
```

Anexo 5 - Interface Socket - net.c