UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Bacharelado em Engenharia Mecatrônica
Julio Cesar Lopez de Souza

SUPORTE A GPIO NO SISTEMA OPERACIONAL NKE

Uberlândia, MG
2021

Julio Cesar Lopez de Souza

SUPORTE A GPIO NO SISTEMA OPERACIONAL NKE

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de Bacharel em Engenharia Mecatrônica, do Curso de Engenharia Mecatrônica da Faculdade de Engenharia Mecânica.

Orientador: Prof. Dr Rivalino Matias Júnior

Uberlândia, MG
2021

RESUMO

Este trabalho consiste na descrição da criação da comunicação com a *General Purpose Input/Output* (GPIO) da placa Beaglebone Black utilizando o NKE, que é um sistema operacional embarcado criado com o intuito de ser utilizado no aprendizado de disciplinas de sistemas operacionais e embarcados. Seu código é majoritariamente escrito em linguagem C e sua execução ocorre utilizando microprocessadores ARM. Este sistema operacional funciona como um *nanokernel*, possuindo um núcleo (*kernel*) minimalista.

Sistemas embarcados são utilizados nos mais diversos aparelhos presentes no nosso dia a dia, além disso, os sistemas embarcados não disponibilizam de uma forma na qual o usuário final possa modificar parâmetros de execução da aplicação do sistema por código. Assim, as formas de o usuário final modificar parâmetros utilizados na execução da aplicação do sistema embarcado são realizadas através de interfaces de comunicação entre hardware e software, por exemplo, através de um teclado que permite definir o tempo de aquecimento de um micro-ondas. Uma das formas mais básicas de o usuário interferir no funcionamento de um sistema embarcado já em execução é utilizando a GPIO.

Desta forma, a implementação da comunicação com a GPIO se torna uma funcionalidade muito importante de ser adicionada no sistema operacional NKE, tendo em vista que o seu uso didático pode se tornar mais abrangente, com a criação de sistemas que interagem com os mais diversos tipos de componentes. Dada a natureza deste trabalho, para seu desenvolvimento foi necessário passar por todo o fluxo de processamento, partindo da criação das estruturas de comunicação com o hardware através da linguagem assembly, até a programação de funções que possam ser utilizadas por programadores para gerenciar a GPIO. Neste trabalho também foi criado um modelo de utilização destas funções através de um estudo de caso onde é possível verificar o seu funcionamento, através da simulação de um sistema de automação residencial.

Palavras-chave: NKE. GPIO. Sistemas operacionais. *Nanokernel*.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização

Atualmente, a utilização de sistemas embarcados está presente nas mais diversas áreas tais como: telefones celulares, equipamentos médicos, automóveis, equipamentos militares, aparelhos domésticos, entre outros. A automatização de diversos sistemas se tornou possível graças aos sistemas embarcados, permitindo a criação de sistemas de alta complexidade e trazendo maior praticidade no dia a dia das pessoas.

Desta forma, a criação de placas cada vez mais simples com objetivo de diminuição de custos para ter maior abrangência se tornou recorrente, o que gera a necessidade de sistemas operacionais mais otimizados para diminuir a utilização de recursos pelo sistema operacional em um sistema embarcado. Assim, a utilização de sistemas operacionais monolíticos, onde o sistema operacional é um "bloco maciço" de código sem limite de acesso à memória e aos recursos do hardware (Maziero, 2019), se tornou uma alternativa menos interessante.

Entre os diversos tipos de núcleo de sistemas operacionais que foram propostos para otimizar o processamento, tem-se por exemplo o *kernel* modular, o microkernel e também o *nanokernel*. O *kernel* modular divide as funcionalidades do sistema operacional em módulos, permitindo a utilização apenas dos módulos específicos para as funcionalidades que são necessárias para o funcionamento da aplicação que irá rodar no sistema embarcado. O microkernel remove do *kernel* a maior parte do código de alto nível, passando estas funcionalidades para o espaço do usuário na forma de serviços, assim, a estrutura do microkernel fica apenas com código de baixo nível (para realizar a comunicação com o hardware) e algumas estruturas mais básicas, esta divisão em serviços também facilita a modularização do *kernel*, já que cada serviço pode ser desenvolvido de forma independente dos outros (Maziero, 2019). O *Nanokernel* tem como características o fato de ter um baixo volume de código por parte do *kernel*, sendo considerado muitas vezes como um microkernel ainda menor (OSDev.org, 2021), além de também possuir uma camada onde se comunica diretamente com o hardware, permitindo menor tempo de resposta por parte do software (Herder, 2005).

O sistema operacional utilizado neste trabalho é o NKE, um sistema operacional embarcado, implementado utilizando o conceito de *Nanokernel*. O NKE é um projeto que foi iniciado com o objetivo de estudo para que os alunos envolvidos pudessem desenvolver e entender o funcionamento de um *nanokernel*. Sua simplicidade e modularidade possibilitam ao estudante, gradativamente, construir seu próprio sistema operacional, a partir de um código base mínimo já existente (COSTA, 2014).

Uma das funcionalidades que ainda não havia sido implementada no NKE consiste na comunicação com a GPIO (*General Purpose Input/Output*), que permite a interação entre software e hardware. Sendo um recurso muito importante para a implementação de projetos embarcados, esta nova funcionalidade se mostrou necessária para o contínuo avanço do NKE.

1.2 Objetivos

1.2.1 Objetivos Gerais

O objetivo deste trabalho consiste na implementação da comunicação GPIO (*General Purpose Input/Output*) utilizando o NKE na Beaglebone Black.

1.2.2 Objetivos Específicos

- Desenvolver um device driver para suporte à controladora de GPIO na Beaglebone Black.
- Estabelecer uma interface GPIO via *system calls* no NKE.
- Realizar um estudo de caso de controle via GPIO usando o NKE.

1.3 Justificativa

A demanda por dispositivos inteligentes e soluções especializadas que podem fornecer resultados eficazes para os problemas diários tornou o uso de microprocessadores e sistemas embarcados uma área importante da computação. Portanto, a busca por sistemas operacionais com diferentes tecnologias empregadas, que podem oferecer suporte a novos dispositivos é crescente.

Nesta situação, este trabalho visa a implementação de uma das ferramentas mais importantes dos sistemas embarcados, a GPIO (*General Purpose Input/Output*), em um sistema operacional mais otimizado como o NKE.

Sendo que as GPIOs são utilizadas para fornecer uma interface entre os periféricos e a placa utilizada, permitindo a criação de sistemas automatizados mais robustos, com respostas visíveis fora do ambiente computacional. A utilização dessa interface em um *nanokernel* como o NKE permite uma maior abstração do funcionamento dessa comunicação para o programador, sendo que o desenvolvedor do sistema não precisa ter conhecimentos aprofundados da interação entre o sistema operacional e as portas GPIO para criar aplicações que se comunicam com os dispositivos que são conectados às portas GPIO.

Um exemplo de utilização da GPIO pode ser verificado em um sistema básico de alarme onde um sensor de presença envia um sinal para a placa que então pode tratar este sinal de diversas formas na aplicação principal, acionando um alarme ou um LED. Utilizando as funcionalidades do NKE criadas neste trabalho, o programador pode desenvolver todas estas interações sem que precise conhecer as chamadas em assembly e interações necessárias com a placa.

A placa Beaglebone Black, utilizada neste trabalho, foi utilizada pois é uma placa que possui grande aplicabilidade, tendo em vista que se trata de uma placa mais robusta e pode executar processamentos mais dispendiosos, permitindo a utilização de diferentes interfaces, que

podem entrar como futuras melhorias no próprio NKE. A placa utiliza um processador ARM, um tipo de processador muito utilizado em sistemas embarcados, devido principalmente ao seu baixo consumo de recursos.

1.4 Metodologia

Este trabalho se dividiu em 3 etapas; (i) estudo da estrutura do NKE e da placa Beaglebone Black; (ii) implementação da comunicação com a GPIO da placa; (iii) estudo de caso, utilizando as funcionalidades implementadas.

1.4.1 Estudo da estrutura do NKE e da placa Beaglebone Black

A primeira fase consiste no estudo dos fluxos de processamento do NKE, para avaliar-se como pode ser estruturada a comunicação com a GPIO dentro do *kernel*, além de um estudo detalhado do hardware da placa Beaglebone Black possibilitando a verificação de como essa comunicação é realizada.

1.4.2 Implementação da comunicação com a GPIO da Beaglebone Black

Com o conhecimento obtido da etapa anterior foi possível criar o fluxo de processamento, utilizando a estrutura do NKE como base para implementar uma comunicação funcional com a GPIO da placa. Devido à comunicação direta com o hardware, o código foi desenvolvido tanto em assembly, para lidar com a controladora GPIO da placa, quanto em C para que fossem criadas as chamadas de sistema (*system calls*) que permitem que o controle destas GPIOs seja feito pelo programador.

1.4.3 Estudo de caso

Por fim tem-se o desenvolvimento de um estudo de caso, onde pode-se verificar o funcionamento do sistema implementado, utilizando um código escrito em C para realizar a interação entre os pinos GPIO que devem ser conectados com diversos componentes eletrônicos, permitindo a criação de um sistema que simula uma aplicação real desta nova funcionalidade.

1.5 Estrutura do Trabalho

Este trabalho foi organizado em cinco capítulos, sendo eles Introdução, Fundamentação teórica, Desenvolvimento da comunicação com a GPIO, Estudo de caso e Conclusão.

O primeiro capítulo tem como objetivo contextualizar o que foi feito, além de mostrar os objetivos e as justificativas que motivaram o seu desenvolvimento.

No segundo capítulo, dedicado à Fundamentação teórica, têm-se os conhecimentos que foram necessários para que fosse possível o desenvolvimento do trabalho, passando por

detalhes importantes sobre a placa utilizada até informações relevantes do fluxo de processamento do NKE.

O terceiro capítulo descreve todas as alterações e implementações que foram necessárias durante o desenvolvimento da comunicação com a GPIO, passando por diversos detalhes de seu fluxo, desde o código assembly até as chamadas de usuário, exibindo como e porque foram realizadas determinadas lógicas de código.

O quarto capítulo apresenta a implementação de um sistema que utiliza as funções criadas no capítulo 3. Neste capítulo é possível verificar todos os detalhes da criação do código, utilizando as chamadas de usuário, também são mostrados todos os componentes de hardware necessários para a montagem do sistema, e por fim tem-se a montagem e teste do sistema conforme a modelagem, analisando-se então seus resultados.

O último capítulo apresenta as conclusões obtidas com o desenvolvimento deste trabalho, comentando detalhes do processo de criação, além de considerações sobre futuras melhorias e novas implementações para o NKE. Após este capítulo o trabalho é finalizado com as referências bibliográficas e os anexos.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são abordados os detalhes associados aos aspectos teóricos utilizados para o desenvolvimento do projeto.

2.1 Sistemas Embarcados

Também conhecidos como sistemas embutidos, os sistemas embarcados tratam-se de sistemas que, utilizando de microprocessadores, permitem a execução de tarefas mais específicas, tendo suas funcionalidades dedicadas a um dispositivo ou sistema no qual ele controla. Sendo que, de forma geral, seu propósito não pode ser alterado ao menos que o mesmo seja reprogramado (CUNHA, 2007). De acordo com EDWARDS et al. (1997) e como foi citado por Silva (2014), "Sistemas que usam um computador para realizar uma função específica, mas não são utilizados ou percebidos como um computador, são geralmente conhecidos como sistemas Embarcados".

Devido à sua aplicabilidade mais específica, estes sistemas podem ter diversas otimizações de tamanho, custo ou até mesmo de recursos computacionais disponíveis, tais como memória, armazenamento, etc (BRAND, 2016). Por causa dessa limitação de recursos, o software presente em um sistema embarcado deve ser mais otimizado para lidar melhor com o uso destes recursos (Freitas, 2013).

De maneira geral, nos sistemas embarcados os usuários finais não terão acesso ao programa que foi embutido no sistema, porém podem interagir com o equipamento através de interfaces como teclados, botões, etc (CUNHA, 2007). Desta forma, algumas das funcionalidades mais importantes em um sistema operacional voltado à utilização em sistemas embarcados é a capacidade de interação com o usuário mesmo sem interface gráfica, entre as diversas formas de realizar esta comunicação a mais comum é através da GPIO.

2.1.1 Arquitetura ARM e a Beagleboard

Com o crescimento da utilização de sistemas embarcados, a necessidade de microprocessadores mais otimizados se tornou cada vez mais presente. Nesta situação, uma das arquiteturas mais bem-sucedidas no mercado é a arquitetura ARM, que utilizando da arquitetura RISC (*Reduced Instruction Set Computer*) possui alto desempenho, com baixo consumo de recursos, além de um menor custo e menor aquecimento, se tornando um tipo de arquitetura perfeita para dispositivos portáteis e sistemas embarcados.

A placa utilizada neste trabalho é a Beaglebone Black, uma placa que utiliza da arquitetura ARM, criada pela Texas Instruments e faz parte de uma serie de placas da linha Beagleboard. As placas da linha Beagleboard são classificadas como placas de hardware livre, a primeira versão de placa da Beagleboard foi lançada em 2008, e com sua crescente utilização, diversas novas versões da placa foram lançadas ao longo dos anos. Sendo que em 2013 foi lançada a Beaglebone Black, que utiliza o microprocessador AM3358BZCZ100 (Texas Instruments,

2011a) com arquitetura ARM Cortex-A8 de 32bits, a placa possui 512MB de memória RAM e clock do processador de 1GHz, quando essa placa foi lançada ela possuia 2GB de memória flash e nas versões mais atuais a placa passou a ter 4GB de memória flash, isso permitiu que a placa passasse a sair de fábrica com o sistema operacional Linux já instalado.

2.1.2 GPIO (*General Purpose Input/Output*)

Os pinos conhecidos como GPIO são portas que podem ser programadas como entrada ou saída, servindo de interface entre o microprocessador e diferentes periféricos. As GPIOs possuem um diverso número de aplicações, podendo desde acender um LED com a saída de tensão direto da placa, como para controlar sistemas de alta potência quando associados com mecanismos como relés, que permitem uma saída de tensão muito maior que o fornecido pela própria GPIO.

A placa Beaglebone Black possui 2 conectores de expansão (P8 e P9) com 46 pinos em cada conector, possuindo assim um total de 92 pinos, sendo que 65 destes pinos podem funcionar como pinos de entrada/saída digital(GPIO), na Figura 1 pode-se verificar a distribuição destes conectores na placa da Beaglebone Black (Beagleboard.org, 2021).

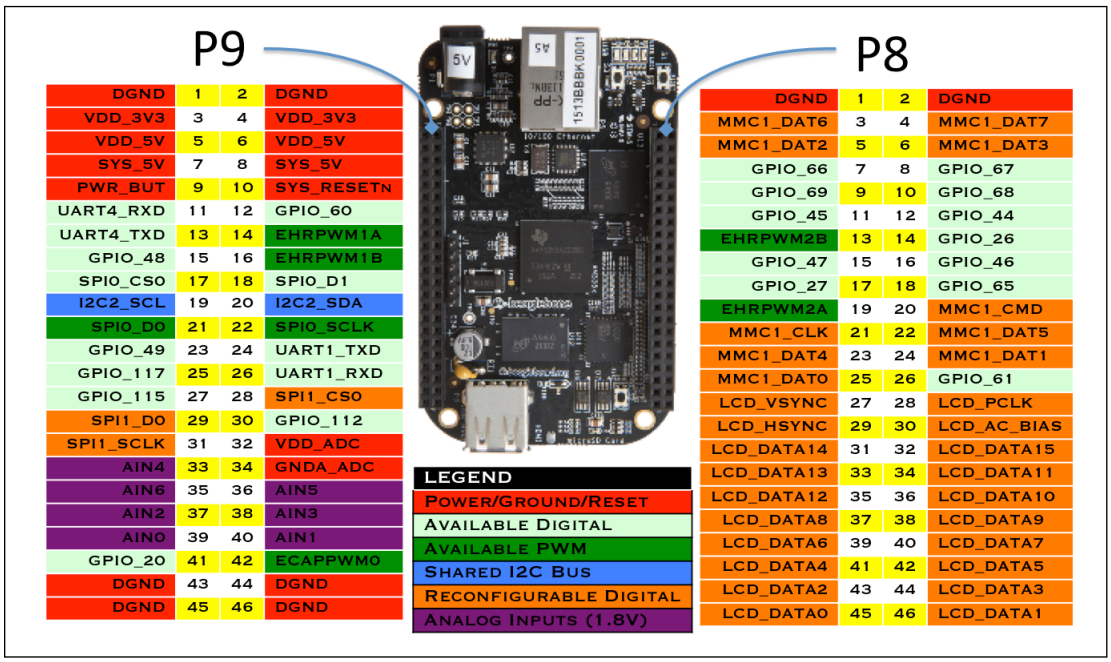

Figura 1 – Distribuição dos pinos nos conectores de expansão P8 e P9.
Fonte: (Beagleboard.org, 2021)

As GPIOs nesta placa são divididas em 4 controladoras, sendo estas: GPIO0, GPIO1, GPIO2 e GPIO3. Cada controladora é responsável por 32 GPIOs, de forma que estas controladoras possuem um mapa de memória cada uma, onde cada endereço neste mapa de memória atua com uma funcionalidade para as GPIOs (Ex: Modificar estado do pino) e cada bit de uma palavra de 32bits deste endereço de memória controla um pino GPIO pelo qual aquela controladora é responsável. Entretanto, nem todas as saídas GPIO são de livre utilização pelo programador, sendo que algumas delas são utilizadas internamente pelo sistema da placa.

Tomando como exemplo a GPIO_45 (pino 11 do conector P8), esta GPIO também pode ser escrita como GPIO1_13, ou seja, o 13º bit da controladora GPIO1, esta conversão de nomenclatura pode ser facilmente obtida a partir do fato de que cada controladora é responsável por 32 GPIOs, conforme a Equação 1.

$$(N\_controladora) * 32 + (Desloc\_GPIO) = (N\_final\_GPIO) \quad (1)$$

Sendo que *N_controladora* é o número referente à controladora, *Desloc_GPIO* é o deslocamento em bits da GPIO e *N_final_GPIO* é o valor final da GPIO. Então, de acordo com a Equação 1, para o pino GPIO1_13, o valor de *N_controladora* é 1 (controladora GPIO1) e o valor de *Desloc_GPIO* é 13 (13º bit), obtendo assim 1*32+13 = 45 que é o valor final de *N_final_GPIO* (GPIO_45).

Assim, cada controladora tem um endereço de memória base, a partir do qual diferentes deslocamentos (*offsets*) formam um endereço final de um registrador, que representa uma palavra de 32 bits, que possui a finalidade de executar uma ação para as GPIOs controladas. Entre estas ações as principais utilizadas neste trabalho são: Ativação da GPIO como entrada/saída, configurar o sinal de saída com valor alto ou baixo e ler o sinal de entrada que está chegando na GPIO.

2.2 Nanokernel

Tendo em vista o objetivo de obter respostas mais rápidas e com menos consumo de recursos para a placa, o desenvolvimento de sistemas operacionais cada vez menores e simplificados se tornou mais recorrente. Dentre estes, o *nanokernel* tem como característica trazer a menor quantidade possível de código por parte do *kernel*, ou seja, menos código sendo executado em modo supervisor (OSDev.org, 2021). Assim, este tipo de sistema operacional deve possuir formas de comunicação mais diretas com o hardware, além de fornecer ao usuário funções que permitam o chaveamento de modo de usuário para o modo supervisor, funções conhecidas como chamadas de sistema (*system calls*).

2.3 NKE

O NKE é um sistema operacional estruturado como um *nanokernel* utilizado no ensino prático de sistemas operacionais. Levando em consideração o número reduzido de funcionalidades presentes em um *nanokernel*, sua estrutura apresenta uma organização conveniente para o ensino prático de sistemas operacionais. Sendo que, ao possuir uma quantidade de código bem reduzida, seu aprendizado se torna mais fácil do que um *kernel* monolítico por exemplo (COSTA, 2014).

Com um conjunto mínimo de funcionalidades, o NKE possui os níveis lógicos de *kernel* e de *usuário*. Os serviços implementados no NKE são executados no nível de *kernel* e podem ser acessados pelo nível de usuário através de chamadas de sistema (COSTA, 2014). O NKE tem seu código escrito majoritariamente na linguagem C, com algumas funções em assembly que são utilizadas principalmente para lidar com os registradores do processador (MURLIKY, 2014).

2.3.1 Chamadas de sistema no NKE

As chamadas de sistema são uma interface que permite a comunicação das aplicações do usuário com o sistema operacional, trazendo uma interface mais amigável e objetiva para o programador. De acordo com Maziero (2019), “Os sistemas operacionais definem chamadas de sistema para todas as operações envolvendo o acesso a recursos de baixo nível (periféricos, arquivos, alocação de memória, etc.) ou abstrações lógicas (criação e finalização de tarefas, operadores de sincronização e comunicação, etc.)”. As chamadas de sistema são oferecidas para o usuário através de funções em nível de usuário, que organizam os parâmetros e invocam as chamadas, retornando seus resultados para a aplicação após finalizarem.

Particularmente no NKE, esta comunicação se dá através de funções disponíveis ao usuário, chamadas de *user calls*, que montam uma estrutura de dados com parâmetros esperados pelo *kernel* e então realizam o chaveamento para o modo supervisor. Neste modo, a informação recebida do usuário será então colocada na fila para processamento como uma chamada de sistema, onde o *kernel* irá tratar e então processar os dados recebidos, comunicando-se com o hardware através do código em assembly, retornando então para o modo de usuário quando termina a chamada.

As estruturas criadas para comunicação entre o modo usuário e modo *kernel* é a estrutura *Parameters*, essa estrutura possui cinco parâmetros *CallNumer*, *p0*, *p1*, *p2* e *p3*. Sendo que *CallNumer* é o número identificador da chamada de sistema, permitindo que o *kernel* trate as entradas dessa estrutura de forma correta, de acordo com a *system call* realizada, *p0*, *p1*, *p2* e *p3* são ponteiros para as variáveis necessárias para o processamento da chamada de sistema. Após a montagem da estrutura *Parameters*, o chaveamento para o modo *kernel* é feito utilizando a função *SWI*, que realiza uma chamada em assembly que passa o controle do sistema para o *kernel*.

As chamadas de sistema já implementadas no NKE são:

- *Criação de Tarefas* - Que permite a criação de *threads* que podem ser definidas como novas tarefas através da chamada *TaskCreate(int *ID,void (*funcao)())*. Sendo que o *ID* recebe o número de identificação da tarefa e a *funcao* é dada por um ponteiro para a função que deve ser executada na *thread*.

- *Inicialização de Tarefas* - Inicia as *threads* criadas através da chamada *TaskCreate*, utilizando a chamada *Start*(SCHEDULER), sendo que a fila de execução é organizada através do escalonamento definido pelo parâmetro SCHEDULER.

- *Sincronização de Tarefas* - Permite a utilização de mecanismos de sincronização entre as *threads* utilizando semáforos, através das funções *seminit(sem_t *semaforo, int ValorInicial)*, *semwait(sem_t *semaforo)* e *sempost(sem_t *semaforo)*. Todas estas funções recebem um ponteiro com a estrutura que identifica o semáforo utilizado, a função *seminit* inicia o ponteiro com o valor passado em *ValorInicial*, a função *sempost* libera o semáforo e a função *semwait* o bloqueia, até que seja liberado com *sempost*.

- *Gerenciamento de tempo* - Permite a temporização de *threads*,passando-as para um estado de espera por um determinado tempo, utilizando as chamadas *Sleep(int time)*, que bloqueia a thread por *time* segundos, e *mSleep(int time)*, que bloqueia a thread por *time* milissegundos.

2.3.2 Comunicação em Assembly

Para realizar a comunicação entre o *kernel* e o hardware para controle das GPIOs, foi criado um arquivo assembly com diversas funções que têm como objetivo gerenciar todas as funcionalidades relativas à GPIO, estas funções são executadas no fluxo de processamento das chamadas de sistema pelo *kernel*. De forma geral, as funções presentes neste arquivo recebem o endereço base da controladora GPIO responsável por aquele pino e o número de bits de deslocamento correspondente àquele pino. Realiza-se então o deslocamento relativo à funcionalidade que se deseja realizar e, por fim, aplica-se um deslocamento em bits relativo à posição (de 0 à 31 bits) daquela GPIO com base na controladora, nesta posição de memória é feita a leitura ou escrita deste bit (de acordo com a funcionalidade).

No caso de ser uma chamada que realiza uma ativação ou escrita, o código assembly irá configurar o valor do bit na posição definida, o que irá realizar a ação correspondente para o pino selecionado, retornando então para o modo usuário. Caso contrário, se for uma chamada de leitura, o código irá verificar se aquele bit possui valor alto ou baixo, e retorna esse valor à variável referenciada pelo usuário, dai então volta ao modo usuário.

De acordo com o manual do processador da Beaglebone Black (Texas Instruments, 2011b), todas as controladoras da GPIO seguem o mesmo padrão de configuração. Assim, para que seja possível acessar as informações de um pino GPIO, precisa-se do endereço base

da controladora, do deslocamento em bits referente àquele pino na controladora e do *offset* correspondente à funcionalidade que se deseja utilizar (mudar o modo de operação, mudar sinal de saída, ler sinal de entrada, etc).

2.4 Utilização da GPIO

A aplicabilidade da GPIO utilizando sistemas embarcados atualmente está presente nas mais diversas áreas, sendo utilizada principalmente em sistemas automatizados, onde a placa realiza o controle de diversos componentes permitindo o funcionamento de um sistema de forma autônoma.

Um exemplo deste tipo de sistema é o caso de um sistema de automação residencial, neste tipo de sistema tem-se uma placa de sistema embarcado programada para controlar diversos aspectos de uma residência, tais como acender e desligar luzes, controle de som ambiente, controle de climatização, gerenciamento de alarmes, podendo também ser utilizado para abrir e fechar portões, além de diversas outras aplicações.

O controle destes sistemas normalmente é feito através de uma interface onde o usuário pode controlar todas estas funcionalidades em um mesmo lugar, por exemplo, através de um aplicativo de celular, que se comunica com a placa por uma conexão de rede utilizando um protocolo cliente/servidor, sendo que para isso a placa trabalha como um servidor, conectada na rede e o software de controle envia sinais através da GPIO permitindo que seja feito o controle de diversos dispositivos de uma residência de forma remota pelo usuário (CRUZ; LISBOA, 2014).

No caso do sistema desenvolvido neste trabalho, o sistema operacional não tem suporte à conexão e comunicação com a rede, entretanto, é possível simular um sistema de automação residencial utilizando a própria GPIO como interface de entrada, utilizando botões por exemplo, para ligar e desligar determinados sistemas. A utilização de sensores também permite que seja feita uma simulação de um sistema de alarme, que pode ser executado automaticamente pelo sistema operacional.

3 DESENVOLVIMENTO DA COMUNICAÇÃO COM A GPIO

3.1 Utilização do NKE na Beaglebone Black

Primeiramente, para que seja estabelecida a comunicação com a placa Beaglebone Black e para visualizar as respostas de impressões por exemplo, foi utilizado de um cabo conversor TTL-USB, que deve ser conectado na porta serial da placa e a outra extremidade deve ser conectada à porta USB do computador, como pode-se ver na Figura 2. Para que seja possível receber as mensagens enviadas da placa, é necessário abrir o Terminal do Linux, e então utilizar o comando *screen* ou *minicom* que simulam o terminal da placa, permitindo que seja feita a leitura da porta USB. Com isso, é possível verificar mensagens geradas pelo código em execução na placa, fator necessário para o desenvolvimento, já que auxilia quando é necessário encontrar problemas na execução do código do projeto.

Quando a placa é iniciada, automaticamente será feito o *boot* e então será iniciada uma distribuição do Linux, que já vem pré instalada com a Beaglebone Black. Para que seja utilizado o NKE como sistema operacional, é preciso compilar o código do projeto, pra isso, com o compilador arm-none-eabi-gcc já instalado no computador, na pasta do projeto existe um arquivo Makefile que permite a compilação do projeto ao utilizar o comando "make", gerando os objetos necessários para a compilação e também o executável do *kernel*.

Desta compilação é utilizado o arquivo ".bin" gerado, que deve ser colocado em um cartão de memória micro SD, este cartão deve conter alguns outros arquivos que permitem que seja realizado o *boot* da imagem na placa. Assim, os arquivos que devem estar presentes no cartão de memória micro SD, para o funcionamento do sistema operacional NKE na Beaglebone Black são:

- *my_kernel.bin* - arquivo gerado na compilação do NKE. Arquivo que contém o *kernel* que será executado na placa.
- *u-boot.img* - arquivo necessário para execução do *boot* no do sistema operacional NKE na placa.
- *uEnv.txt* - arquivo utilizado para configuração do *boot* da placa, que define parâmetros da placa para execução do *kernel* desenvolvido (sunxi.org, 2021).

Este cartão de memória deve ser inserido na Beaglebone Black na entrada especificada para tal e, ao conectar a placa em uma fonte de energia, deve ser mantido pressionado o botão de *boot* presente na placa, para que a inicialização seja feita a partir do cartão e não do sistema operacional vindo de fábrica.

Com estes passos realizados pode-se então verificar o funcionamento correto do sistema operacional NKE na Beaglebone Black e visualizar suas respostas através do terminal do Linux no computador.

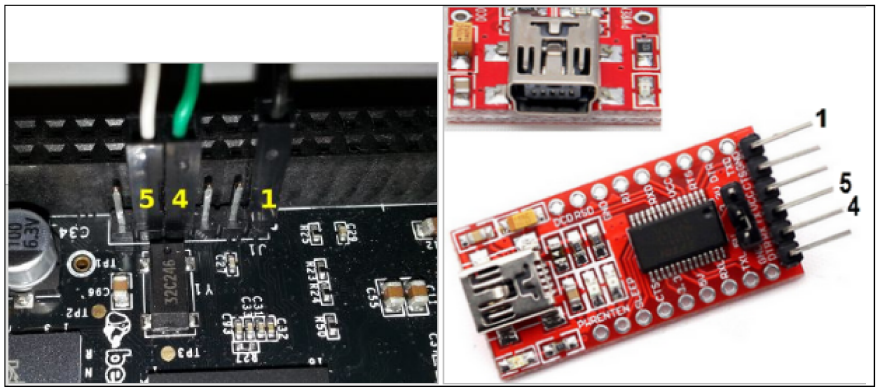

Figura 2 – Porta Serial da Beaglebone Black e conversor TTL-USB
Fonte: (Burjaili; Oliveira, 2013)

3.2 Fluxo de chamadas no NKE

Para auxiliar o entendimento do fluxo de execução de chamadas criadas para a comunicação com a GPIO, a Figura 3 exibe um fluxograma da sequência de execução que ocorre dentro do NKE ao realizar uma requisição de manipulação de alguma GPIO.

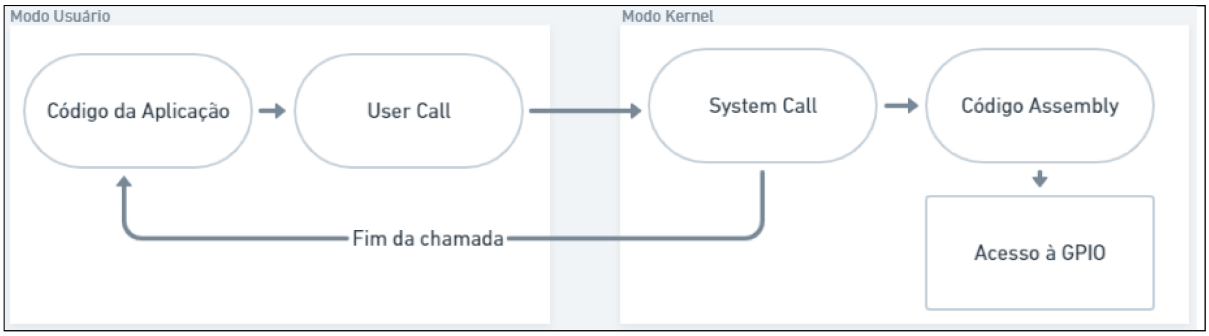

Figura 3 – Fluxograma das chamadas criadas para comunicação com a GPIO

Assim, existem dois modos de operação que são executados no NKE, o *modo usuário* e o *modo kernel*. No *modo usuário*, o "Código da Aplicação" possui a estruturação da lógica que deve ser criada pelo programador para interagir entre diversas GPIOs, o acesso à essas GPIOs é feito utilizando funções presentes na "User Call", onde as informações que precisam ser acessadas (fornecidas pela aplicação) são distribuídas em uma estrutura já conhecida pelo *kernel* e é feito então o chaveamento do *modo usuário* para o *modo kernel*.

No *modo kernel*, as estruturas criadas na "User Call" entram na fila de processamento e quando executadas são realizadas as chamadas de sistema ("System Call") correspondentes às estruturas que foram recebidas. Nas funções de chamada de sistema relativas à GPIO é feito o acesso à funções do "Código Assembly" que permitem o acesso aos registradores da placa, realizando então o "Acesso à GPIO". Quando a função de uma chamada de sistema finaliza o processamento, o código continua sua execução na aplicação do *modo usuário* a partir da linha seguinte à chamada de usuário realizada.

3.3 Comunicação com a GPIO

A comunicação com a GPIO no NKE foi desenvolvida utilizando diretamente os registradores de acesso das controladoras GPIO, conforme especificado no manual do processador da placa (Texas Instruments, 2011b). Para tal foi criado um código em assembly que permite o acesso às posições de memória necessárias para realização do controle das GPIOs.

Para realizar este controle, são utilizadas cinco diferentes chamadas para a placa, que são na realidade deslocamentos (*offsets*) a partir do registrador base de uma dada controladora. Estas chamadas foram utilizadas em diferentes funções assembly no código para que dinamicamente pudesse ser feita a conversão do endereço base da registradora em um registrador referente à função que deve ser realizada. Também recebe-se qual o pino em que será utilizado, representado pelo bit relativo à registradora manipulada. Assim, conforme o manual da placa, foram implementados os seguintes *offsets*:

- `GPIO_OE` (0x134) – Responsável por modificar o modo de funcionamento da GPIO (entrada ou saída).
- `GPIO_CLEARDATAOUT` (0x190) – Responsável por limpar o valor de saída da GPIO (mudando o sinal de saída para baixo).
- `GPIO_SETDATAOUT` (0x194) – Responsável por setar o valor de saída na GPIO (mudando o sinal de saída para alto ou baixo).
- `GPIO_DATAOUT` (0x13C) – Responsável pela leitura do sinal de saída da GPIO (verificando sinal alto ou baixo naquele pino).
- `GPIO_DATAIN` (0x138) – Responsável pela leitura do sinal de entrada da GPIO (verificando sinal alto ou baixo naquele pino)

De forma que, tomando como exemplo, se for necessário ativar o pino GPIO1_12 como pino de saída, é preciso pegar o registrador base da GPIO1 que é dado pelo endereço 0x4804c000, aplicar o deslocamento referente à modificação do modo de funcionamento, que é dado pela GPIO_OE com o valor 0x134, obtendo então o registrador relativo a essa função na GPIO1 pelo endereço 0x4804c134. Deve-se então carregar neste registrador o valor relativo à modificação para sinal de saída (valor zero) no 12º bit do mesmo.

De maneira geral, é assim que é feita a interação com as controladoras para gerenciar o controle das GPIOs, sendo que, no caso das funções de leitura, ao invés de carregar um valor na memória, é realizada a leitura do valor do registrador. Entretanto, diversos cuidados tiveram de ser tomados para garantir o funcionamento de um sistema mais dinâmico e que permitisse o funcionamento de mais de um pino por controladora simultaneamente, sem interferência entre eles.

3.3.1 Código Assembly

Conforme citado anteriormente, para gerir a comunicação direta com as controladoras através dos registradores, foram utilizadas funções em assembly, o que permitiu a leitura e escrita nestes endereços de memória de forma mais direta. Para tal, foi então criada um arquivo assembly chamado “gpiohandler.S”, neste arquivo tem-se as seguintes funções:

- InitGpioOutput – responsável por mudar o modo de um pino para saída, utilizando o GPIO_OE.
- InitGpioInput – responsável por mudar o modo de um pino para entrada, utilizando o GPIO_OE.
- SetGpioOutOn – responsável por mudar o estado de um pino que seja de saída para gerar sinal alto (um), utilizando o GPIO_SETDATAOUT.
- SetGpioOutOff – responsável por mudar o estado de um pino que seja de saída para gerar sinal baixo (zero), utilizando o GPIO_CLEARDATAOUT.
- GetGpioOutValue – responsável pela leitura do sinal de um pino de saída e retorná-lo, utilizando o GPIO_DATAOUT.
- GetGpioInValue – responsável pela leitura do sinal de um pino de entrada e retorná-lo, utilizando o GPIO_DATAIN.

De maneira geral, todas estas funções recebem 2 parâmetros, sendo o primeiro o endereço base da controladora (Ex: 0x4804c000 para GPIO1), e o segundo o valor correspondente ao deslocamento em bits do pino que se deseja alterar (Ex: 1«12 para GPIO1_12). Pela forma de funcionamento das funções em assembly na arquitetura ARM (Furber, 2000), estes parâmetros são recebidos pelas funções nos registradores da memória r0 e r1.

Assim, com exceção das funções responsáveis pela leitura (GetGpioInValue e GetGpioOutValue) dos valores da registradora, as outras 4 funções tratam da leitura e escrita nos endereços das controladoras, possuindo muitas semelhanças no seu código. Como exemplo, o Código 1 apresenta o código da função InitGpioOutput. Todas as funções assembly criadas na “gpiohandler.S” podem ser verificadas no Código 23 no Anexo A.

Código 1 – Código assembly de exemplo da função InitGpioOutput

```
1  InitGpioOutput:
2          stmfd sp!, {r2-r8, lr}
3          .equ GPIO_OE, 0x134
4          MOV          r3, #GPIO_OE
5          ADD          r7,r0,r3
6          SWP          r7,r7,[r8]
7          LDR          r7,[r8]
8
9          ORR          r5,r7,r1
10         SUB          r4,r5,r1
11         STR          r4,[r7]
12
13         ldmfd sp!, {r2-r8,lr}
14         MOV          pc,lr
```

Este código de exemplo possui a mesma estrutura das outras funções de escrita criadas para gerenciar a GPIO, e pode ser dividido em basicamente 3 seções. A primeira corresponde ao início, onde são recebidos os dados e é realizado seu tratamento para que possam ser utilizados para manipulação das informações da GPIO conforme o desejado. A segunda parte é a parte intermediária, onde realmente é feita a manipulação dos dados, acessando os valores da controladora e o reescrevendo-os de acordo com o necessário. A terceira e última parte permite a correta finalização da chamada em assembly, e a continuação do fluxo do código.

Na primeira parte, todas as funções de escrita tem a mesma estrutura, com o diferencial apenas do *offset* que é utilizado, como exemplo, no Código 2 o *offset* utilizado refere-se à GPIO_OE.

Código 2 – Código assembly comum a maioria das funções implementadas

```
1  stmfd sp!, {r2-r8, lr} // salva os registradores
2  .equ GPIO_OE, 0x134
3  MOV  r3, #GPIO_OE
4  ADD  r7,r0,r3
5  SWP  r7,r7,[r8]
6  LDR  r7,[r8]
```

No Código 2, a primeira linha é responsável por salvar os registradores da memória, para que seja possível retornar os registradores ao seu estado inicial antes de retornar da função.

A segunda linha nomeia uma constante com o *offset* referente à função da GPIO que deve ser utilizada, para facilitar a leitura e compreensão do código.

Na terceira linha é iniciada a interação com os registradores dentro da função, utilizando o comando MOV, armazena-se no registrador r3 o valor referente à função que será utilizada. Neste caso, como o endereço é referente ao GPIO_OE, este valor estará armazenado no registrador r3, que está sendo utilizado como variável temporária.

Na quarta linha, é realizada uma soma entre os registradores r0 e r3, e então este valor é armazenado no registrador r7, o registrador r0 neste ponto tem armazenado nele o endereço base da controladora já todas as funções recebem este como primeiro parâmetro. Esta soma realiza o deslocamento para o registrador da função desejada para aquela registradora, assim, se o endereço recebido é o da GPIO1, neste caso, o valor que será armazenado no registrador r7 é 0x4804c134.

A quinta linha realiza a chamada SWP, que permite uma troca de valor e endereço entre até 3 registradores (ARM KEIL, 2021). Isso foi utilizado pois apesar de ter-se o endereço do registrador que será acessado, este endereço precisa estar associado como endereço de memória do registrador para que seja possível carregá-lo e então manipular o registrador da GPIO. Sem este passo o registrador r7 tem armazenado nele o endereço necessário, mas endereço do registrador r7 em si não corresponde à este valor, assim, esta chamada pega o endereço armazenado em r7 e salva no endereço do registrador r8. A última linha desta seção de código é responsável por carregar o registrador que será manipulado, carregando o endereço do registrador r8 no registrador r7.

Seguindo para a seção intermediária do código, são iniciadas as manipulações específicas de cada função, na seção de código presente no Código 3 apresenta os detalhes específicos relativos à função InitGpioOutput:

Código 3 – Código especifico à função InitGpioOutput

```
1 ORR        r5,r7,r1                    //Garante que o bit do pino
     esteja igual à 1 no registrador r7, salvando este valor no
     registrador r5
2 SUB        r4,r5,r1                    //Seta o bit do pino como
     zero e salva no registrador r4
3 STR        r4,[r7]                  //seta o deslocamento(r4) em
     relacao ao r7(base+GPIO_OE)
```

Primeiramente, é importante lembrar que o registrador r1 recebeu o deslocamento em bits da registradora referente ao pino que pretende-se alterar, além disso, o registrador r7 retorna o valor presente no registrador GPIO_OE da registradora especificada quando acessado, ou seja, um conjunto de 32 bits que informa o modo de funcionamento de cada pino da registradora.

Assim, nesta primeira linha, é recebido o deslocamento em bits referente ao pino e aplicado um operador lógico OR entre este deslocamento e os 32 bits da controladora, garantindo que a posição em bits do pino especificado seja um, na segunda linha é realizada uma subtração entre o valor obtido (presente no registrador r5) e o deslocamento presente no registrador r1,

fazendo com que a posição em bits do pino passe a ser zero.

Esta lógica é realizada pois a função InitGpioOutput tem como objetivo setar o pino especificado como pino de saída, ou seja, no bit referente àquele pino deve-se ter valor zero, como não é conhecido o estado do pino no momento que é feita a leitura dos bits da controladora utiliza-se deste artifício lógico para garantir que nenhum dos outros bits será afetado e o bit correspondente ao pino possua o valor desejado. A terceira e última linha, pega o valor final com os valores dos bits de cada pino da controladora e salva no registrador da mesma, alterando o estado do pino recebido.

De forma análoga, a função InitGpioInput deve setar o pino como input, e portanto, o bit correspondente ao seu deslocamento deve possuir valor um. Assim, seu código é praticamente o mesmo da função InitGpioOutput, com exceção da segunda linha, onde é feita a subtração do bit de deslocamento, já que o valor final do bit deve permanecer positivo.

Essa lógica é igual também para as funções SetGpioOutOff e SetGpioOutOn, já que uma acessa o registrador de limpeza de sinal e a outra acessa o registrador de configurar sinal de saída, ou seja, todos tem o mesmo código específico, mudando apenas o valor presente no registrador 7, que é específico de cada tipo de ação que deve ser aplicada na função (dado pelo deslocamento citado anteriormente), como exemplo, o Código 4 mostra essa seção para a função InitGpioInput.

Código 4 – Código especifico das funções InitGpioInput

```
1 ORR          r1,r7,r1
2 STR          r1,[r7]     //seta o deslocamento(r4) em relacao ao r7
     (base+GPIO_OE)
```

Ao final de todas as funções que foram criadas para manipular a GPIO uma mesma seção de código também se faz presente, como pode ser visto no código a seguir, onde na primeira linha os registradores são restaurados para seu estado inicial e então na segunda linha retorna-se à execução da função antes de entrar no assembly.

Código 5 – Código geral que finaliza as funções assembly

```
1 ldmfd sp!, {r2-r8,lr} // restaura os registradores
2 MOV          pc,lr      // Retorna para a funcao anterior
```

Isso finaliza a estrutura geral das funções que realizam a escrita nos registradores das controladoras GPIO. As outras duas funções que são utilizadas realizam apenas a leitura dos dados presentes nestes registradores, e assim, seu código possui algumas diferenças, o Código 6 exibe o código da função GetGpioInValue como exemplo.

Código 6 – Código de modelo para funções de leitura

```
1 GetGpioInValue:
2         stmfd sp!, {r2-r8, lr}
3         .equ GPIO_DATAIN, 0x138
4
5         LDR          r0,[r0,#GPIO_DATAIN]
6         AND          r0,r0,r1
7
8         ldmfd sp!, {r2-r8,lr}
9         MOV          pc,lr
```

Como é possível verificar no Código 6, para funções de leitura, tem-se a mesma estrutura das funções de escrita no início e no final do código, utilizando as chamadas necessárias para guardar e restaurar os registradores, além de declarar a constante relativa à chamada da função (GPIO_DATAIN no exemplo) e finalizar com a linha para retornar ao fluxo de execução anterior. Assim, a principal diferença deste tipo de função para as outras explicadas anteriormente corresponde à seção intermediaria, que neste exemplo podem ser verificadas nas linhas 5 e 6 do código.

Na linha 5 é realizada a leitura direta do registrador base da controladora (presente no registrador r0) com o deslocamento da chamada que deve ser realizada, como nenhuma escrita será feita pode-se realizar a leitura direta do endereço com a chamada LDR passando o deslocamento da chamada junto de r0, atualizando então o valor de r0 para o valor presente no endereço lido.

Na linha 6 é verificado se o valor do bit correspondente ao deslocamento do pino (presente no registrador r1) está ou não configurado no registrador lido, realizando a chamada AND entre o r0 atualizado e r1, atualizando o valor de r0 com o resultado desta chamada para que possa ser retornado pela função. A função GetGpioOutValue tem esta mesma estrutura, porém é utiliza sua chamada específica, utilizando a constante GPI_DATAOUT no lugar de GPIO_DATAIN.

3.3.2 Definições da GPIO

Para realizar as manipulações na GPIO é necessário que sejam definidas algumas constantes, que podem ser utilizadas pelos usuários e são necessárias para o tratamento dentro das chamadas de sistema. Para isso foi criado o cabeçalho “gpio.h”, onde todas estas constantes são criadas, de forma que, 4 tipos de informações são declaradas neste cabeçalho. A primeira corresponde ao endereço base de cada uma das controladoras da GPIO, para que seja utilizado o nome declarado ao invés do endereço puro no código, funcionando de forma mais intuitiva na análise do código.

Código 7 – Definição dos endereços das controladoras

```
#define GPIO0_BASE                          (0x44e07000)
#define GPIO1_BASE                          (0x4804c000)
#define GPIO2_BASE                          (0x481ac000)
#define GPIO3_BASE                          (0x481ae000)
```

Através destas definições pode-se chamar GPIO0_BASE e não seu endereço quando pretende-se utilizar a registradora GPIO0, por exemplo, facilitando o entendimento do código. O segundo e o terceiro tipo de informação correspondem aos estados possíveis para a GPIO e ao tipo de GPIO (modo de operação do pino) aceito pelo *kernel*, que foram declarados utilizando uma enumeração, limitando as informações que o programador poderá inserir nas requisições realizadas. Esta limitação foi realizada para padronizar as informações que podem ser inseridas pelo programador, forçando-o a utilizar apenas valores que são aceitos pelo *kernel*.

Código 8 – Definição dos estados e tipos possíveis para as GPIOs

```
typedef enum
{
    HIGH = 1,
    LOW = 0
}GPIO_STATE;
typedef enum
{
    GPIO_INPUT = 1,
    GPIO_OUTPUT = 0
}GPIO_TYPE;
```

Como pode ser verificado no Código 8, a enumeração GPIO_STATE define os estados possíveis para o pino e a enumeração GPI_TYPE define os modos de operação aceitos para as GPIOs. O último tipo de informação definida corresponde ao pino que será manipulado, utilizando também de uma enumeração, o acesso ao pino pelo usuário pode ser feito por diferentes nomes.

Por exemplo, o pino P8_18 corresponde ao 18º pino do barramento P8, este pino também pode ser visualizado como GPIO2_1 referenciando a controladora à qual ele está associado, ou então também pode ser visto como GPIO_65 quando verifica-se a numeração geral dos pinos. Assim, para dar mais liberdade à nomenclatura desejada pelo programador e limitar os pinos que são aceitos na manipulação pela placa, é utilizada a enumeração GPIOS, que declara todos os pinos que podem ser usados de diferentes formas, o Código 9 exibe parte desta enumeração.

Código 9 – Parte da definição dos pinos disponíveis para serem manipulados

```
typedef enum
{
    ...
    P8_18 = 65,
    GPIO2_1 = 65,

    P8_07 = 66,
    GPIO2_2 = 66,
    ...
}GPIOS;
```

Conforme o Código 9, a declaração dos pinos que podem ser utilizados é feita de 3 formas possíveis, para que o programador possa utilizar a forma que quiser para representar o pino no desenvolvimento de uma aplicação à nível de usuário.

3.3.3 Funções em C a nível de *kernel*

Para que seja possível realizar os tratamentos corretos dos dados para manipulação das GPIOs e utilizar corretamente as funções em assembly criadas dentro do NKE, algumas funções devem ser criadas. Primeiramente, para que as funções presentes na "gpiohandler.S" sejam utilizadas no código C é necessário declará-las no cabeçalho "kernel.h", permitindo o acesso às funções como se fossem parte do código em C, conforme o Código 10.

Código 10 – Declaração das funções assembly no cabeçalho do *kernel*

```
/* gpiohandler.S */
void InitGpioOutput(unsigned int GPIO_BASE, unsigned int
   pin_number_hex) __attribute__ ((__long_call__));
void InitGpioInput(unsigned int GPIO_BASE, unsigned int
   pin_number_hex) __attribute__ ((__long_call__));
void SetGpioOutOn(unsigned int GPIO_BASE, unsigned int
   pin_number_hex) __attribute__ ((__long_call__));
void SetGpioOutOff(unsigned int GPIO_BASE, unsigned int
   pin_number_hex) __attribute__ ((__long_call__));
int GetGpioOutValue(unsigned int GPIO_BASE, unsigned int
   pin_number_hex) __attribute__ ((__long_call__));
int GetGpioInValue(unsigned int GPIO_BASE, unsigned int
   pin_number_hex) __attribute__ ((__long_call__));
```

Através destas declarações se torna possível utilizar estas funções diretamente dentro das chamadas de sistema para manipular as GPIOs conforme necessário. As chamadas de

sistema realizadas no NKE passam por uma verificação do número da chamada (*CallNumber*) que define que tipo de tratamento deve ser feito em uma dada chamada de sistema, assim, para as chamadas da GPIO foram criados 4 novos tipos chamadas: GPIOINIT, GPIOSET, GPIOGET e GPIOGETOUT. Estas chamadas são definidas na enumeração de chamadas de sistema no cabeçalho "syscall.h", o Código 11 apresenta parte desta enumeração.

Código 11 – Parte da enumeração que define as chamadas de sistema

```
enum sys_temCall
{
    ...
        GPIOINIT,
        GPIOSET,
        GPIOGET,
        GPIOGETOUT
};
```

Cada uma destas chamadas é tratada por uma função diferente, estas funções são todas definidas também no cabeçalho "syscall.h" conforme o código Código 12.

Código 12 – Declaração das tratativas das chamadas de sistema para GPIO

```
void sys_gpioinit(unsigned int gpio,unsigned int type);
void sys_gpioset(unsigned int gpio,unsigned int state);
void sys_gpioget(unsigned int gpio,unsigned int* reference);
void sys_gpioget_output(unsigned int gpio,unsigned int* reference
   );
```

Além disso, uma outra função foi criada para auxiliar no tratamento dos dados recebidos na chamada de sistema, a função *sys_getgpiobase*, que recebe o número da GPIO e retorna o endereço base da controladora GPIO correspondente, conforme o Código 13.

Nesta função, o valor recebido na variável "gpio" corresponde à numeração de um pino conforme a enumeração no cabeçalho "gpio.h", ou seja, se por exemplo o pino utilizado for o P8_18, o valor da variável "gpio" será igual a 65. Sendo que cada controladora possui 32 pinos, conforme explicado nas seções anteriores, esta função verifica à qual controladora o pino pertence pegando a parte inteira da divisão da numeração do pino por 32, assim, para gpio=65 tem-se que a controladora seria a controladora GPIO2, já que a parte inteira da divisão 65 por 32 é igual a 2. Esta função foi criada pois todas as tratativas das chamadas de sistema da GPIO recebem o número do pino, mas as funções em assembly recebem o endereço base da controladora, sendo uma conversão que todas as funções de tratamento das chamadas de sistema da GPIO precisam executar.

Código 13 – Função para obter o endereço base da controladora correspondente ao pino

```
unsigned int sys_getgpiobase(unsigned int gpio)
{
        unsigned int gpio_number = (unsigned int)gpio/32;
        switch (gpio_number)
        {
                case 0:
                        return GPIO0_BASE;
                case 1:
                        return GPIO1_BASE;
                case 2:
                        return GPIO2_BASE;
                case 3:
                        return GPIO3_BASE;
                default:
                        return 0;
        }
}
```

Todas as funções de tratamento presentes no Código 12 possuem uma mesma seção de código onde é realizada a validação do endereço base da controladora e então é extraído o número de bits de deslocamento correspondente àquele pino na controladora, conforme o Código 14.

Código 14 – Validação do endereço da controladora e extração do deslocamento do pino

```
unsigned int gpio_base = sys_getgpiobase(gpio);
if (gpio_base != 0)
{
        unsigned int gpio_entry = gpio%32;
        ...
}
```

Analogamente ao que foi feito na função *sys_getgpiobase*, para obter o deslocamento do pino é calculado o resto da divisão do número da gpio por 32, assim, seguindo o exemplo citado anteriormente, se gpio=65 o resto da divisão por 32 será 1, e portanto pode-se verificar que este pino corresponde à GPIO2_1.

Todas as chamadas entretanto possuem suas peculiaridades, sendo que elas chamam diferentes funções do código assembly e têm diferentes objetivos. O Código 15 serve de exemplo para que seja possível verificar como é a estrutura geral de uma destas funções.

Código 15 – Exemplo do código de uma das tratativas das chamadas de sistema

```
void sys_gpioinit(unsigned int gpio,unsigned int type)
{
        unsigned int gpio_base = sys_getgpiobase(gpio);
        if (gpio_base != 0)
        {
                int gpio_entry = gpio%32;
                if (type == (unsigned int)GPIO_INPUT)
                  InitGpioInput(gpio_base,(1<<gpio_entry));
                else
                  InitGpioOutput(gpio_base,(1<<gpio_entry));
        }
}
```

O Código 15 corresponde à função *sys_gpioinit*, que recebe o número da gpio e o tipo, ou modo de operação, que deseja-se iniciar o pino. Este tipo possui apenas dois valores possíveis (GPIO_INPUT e GPIO_OUTPUT) como foi visto no Código 8, assim, quando o pino deve funcionar como pino de entrada função assembly que deve ser chamada é a InitGpioInput e caso seja pino de saída deve ser chamada então a função InitGpioOutput, como pode ser visto na seção específica desta função.

Um outro detalhe que pode ser percebido nesta função é com relação ao deslocamento em bits correspondente ao pino, representado pela variável "gpio_entry", as funções assembly criadas esperam o valor já deslocado dos bits para realizar suas funções, por isso, deve ser realizado um *shift* binário do valor 1 no número de casas correspondente ao gpio_entry, ou seja, se gpio_entry é igual à 2 então o valor que deve ser passado para a função é 100 (valor em binário).

A função *sys_gpioset* funciona de forma semelhante à *sys_gpioinit*, porém em vez de receber o modo de operação do pino ela recebe o estado que o pino deve ser configurado, dado pela variável "state", assim, a seção específica do código dessa função pode ser visualizada no Código 16.

Código 16 – Seção específica do código da função *sys_gpioset*

```
if (state == (unsigned int)HIGH)
        SetGpioOutOn(gpio_base,(1<<gpio_entry));
else
        SetGpioOutOff(gpio_base,(1<<gpio_entry));
```

Nesta função tem-se que apenas 2 estados podem ser definidos para a saída da GPIO, semelhantemente ao que ocorre na função *sys_gpioinit*, sendo, valor HIGH quando o sinal de saída deve ser alto e LOW quando a saída deve possuir sinal baixo. Então esta função permite

que seja acionado um pino de saída com sinal alto através da função assembly SetGpioOutOn ou com sinal baixo utilizando SetGpioOutOff.

As funções *_gpioget* e *sys_gpioget_output* possuem a mesma estrutura, mudando apenas qual função assembly será chamada, enquanto a função *sys_gpioget* chama a função assembly GetGpioInValue, a função *sys_gpioget_output* chama a função assembly GetGpioOutValue. Ambas tratativas de chamadas de sistema recebem o número da gpio e um ponteiro, dado pela variável "reference". Como estas funções têm como objetivo realizar a leitura de um estado de um pino, este ponteiro serve para que este valor seja retornado do modo *kernel* para o modo usuário. No Código 17 tem-se a seção específica da função *sys_gpioget* como exemplo.

Código 17 – Seção específica do código da função *sys_gpioget*

```
1 int valor = 0,gpio_val = (1<<gpio_entry);
2 valor = GetGpioInValue(gpio_base,gpio_val);
3 *reference = valor/gpio_val;
```

Nesta seção de código verifica-se que a principal diferença com as outras tratativas citadas se dá ao receber um valor da chamada assembly realizada e o tratamento deste valor. Ambas as funções de leitura criadas em assembly verificam se o deslocamento de bit correspondente ao pino possui valor 1 ou 0, assim, se o valor for zero a função também retornará zero, senão retornará o próprio deslocamento do pino, por isso na terceira linha do Código 17 é armazenado no valor do ponteiro de referência a divisão entre o valor retornado da função assembly e o deslocamento em binário correspondente ao pino, fazendo com que os únicos valores possíveis na variável de referência seja 1 ou 0.

3.3.4 Funções em C a nível de usuário

As chamadas para manipulação das GPIOs a nível de usuário são criadas através das *usercalls*, nela são definidos alguns parâmetros que devem ser passados para as funções e é montada uma estrutura que pode ser então interpretada a nível de *kernel*. As chamadas para GPIO podem ser verificadas no Código 18.

Código 18 – Declaração das funções que podem ser utilizadas pelos usuários

```
1 void gpiowrite(GPIOS GPIO_name, GPIO_STATE state);
2 void gpioinit(GPIOS GPIO_name, GPIO_TYPE mode);
3 void gpioread(GPIOS GPIO_name, int *value, GPIO_TYPE mode);
4 void gpioreadin(GPIOS GPIO_name, int *value);
```

Estas 4 funções são definidas no cabeçalho "usercall.h", seus parâmetros de entrada possíveis foram limitados para que só recebam informações da GPIO que estão presentes nas enumerações do cabeçalho "gpio.h", para evitar que o programador envie requisições que não podem ser processadas pelo *kernel* ou que não façam sentido para o hardware da placa.

A estrutura utilizada para passar estes parâmetros para o *kernel* é a *Parameters*, que salva primeiramente o *CallNumber* correspondente à chamada de sistema que deve ser feita e então salva os outros parâmetros nas variáveis p0 e p1, para que o tratamento das chamadas de sistema possam ser realizados de maneira correta a nível de *kernel*. No Código 19 pode-se verificar a criação de todas estas funções no arquivo “usercall.c”.

Como é possível observar, cada uma destas funções possui um *CallNumber* específico, que corresponde à chamada de sistema que deverá ser realizada, os parâmetros salvos na estrutura são então passados como referência à função *CallSWI* que passa o controle para o NKE e permite o início da chamada de sistema, onde tem-se todas as tratativas e chamadas às funções assembly citadas nas seções anteriores.

As funções *gpioread* e *gpioreadin* são as funções responsáveis pela leitura dos valores da GPIO, de forma que a variável *value* corresponde ao ponteiro que terá seu valor alterado após a leitura no *kernel*. A princípio a função *gpioread* por si só já basta para tratar qualquer tipo de chamada de leitura, já que ela verifica se a GPIO que será lida é do tipo *input* ou *output*, porém a função *gpioreadin* foi criada para facilitar a criação de códigos para o programador, já que a maior parte das utilizações de leitura de estados do pino ocorrem quando a GPIO está configurada como *input*.

Código 19 – Funções utilizadas para montar a estrutura e realizar as chamadas de usuário

```
void gpioinit(GPIOS GPIO_name, GPIO_TYPE mode)
{
        Parameters arg;
        arg.CallNumber = GPIOINIT;
        arg.p0 = (unsigned char *) (long) GPIO_name;
        arg.p1 = (unsigned char *) (long) mode;
        CallSWI(0, &arg);
}
void gpiowrite(GPIOS GPIO_name, GPIO_STATE state)
{
        Parameters arg;
        arg.CallNumber = GPIOSET;
        arg.p0 = (unsigned char *) (long) GPIO_name;
        arg.p1 = (unsigned char *) (long) state;
        CallSWI(0, &arg);
}
void gpioread(GPIOS GPIO_name, int *value, GPIO_TYPE mode)
{
        Parameters arg;
        if (mode == GPIO_OUTPUT)
                arg.CallNumber = GPIOGETOUT;
        else
                arg.CallNumber = GPIOGET;
        arg.p0 = (unsigned char *) (long) GPIO_name;
        arg.p1 = (unsigned char *) value;
        CallSWI(0, &arg);
}
void gpioreadin(GPIOS GPIO_name, int *value)
{
        Parameters arg;
        arg.CallNumber = GPIOGET;
        arg.p0 = (unsigned char *) (long) GPIO_name;
        arg.p1 = (unsigned char *) value;
        CallSWI(0, &arg);
}
```

4 ESTUDO DE CASO

Para que fosse possível validar o funcionamento correto do fluxo e controle das GPIOs, foi modelado um sistema simplificado de automação residencial. Conforme tratado anteriormente, é possível simular um sistema deste tipo utilizando a própria GPIO como dispositivo de entrada para controle. Neste estudo foram criados 2 sistemas de controle que rodam simultaneamente, um permite a verificação de um modelo completamente automático de alarme e o outro a simulação de uma ativação remota de iluminação.

4.1 Modelagem do circuito

Primeiramente foi necessário a montagem do circuito de simulação de um sistema de automação residencial, de forma que, para isso foram utilizados os seguintes componentes eletrônicos:

- 1 Protoboard
- 1 Sensor de movimento e presença
- 2 LEDs
- 1 Botão
- Resistores
- Jumpers (fios de conexão)
- Beaglebone Black

4.1.1 Detalhes dos componentes

A Protoboard se trata de uma placa com diversos furos, com conexões condutoras verticais e horizontais conectando estes furos. As Protoboards auxiliam na prototipagem de circuitos eletrônicos, permitindo a montagem dos mais diversos circuitos sem a necessidade de montar uma placa de circuito impresso para realizar as conexões. A Figura 4 apresenta uma imagem da Protoboard utilizada no desenvolvimento.

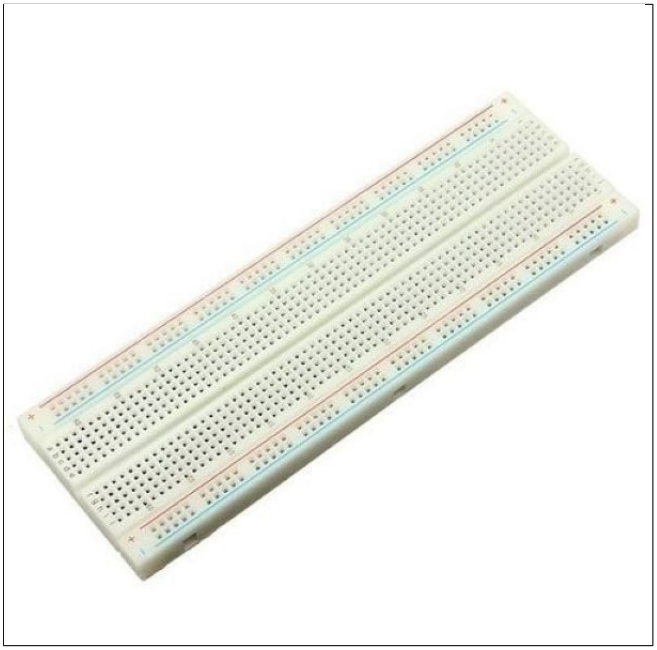

Figura 4 – Protoboard utilizada para montagem do circuito
Fonte: (Filipeflop, 2021)

O sensor de movimento e presença utilizado foi o PIR DYP-ME003, este sensor possui 3 conectores, um para aterramento, um para alimentação do sensor que permite sua operação que pode ser de 4,5-20V e o terceiro conector é o conector de dados, que envia um sinal de 3,3V quando o sensor percebe movimento. Além disso, este sensor possui 2 potenciômetros para ajuste de tempo e sensibilidade. Ajustando o tempo é possível aumentar ou diminuir o tempo que o sensor manterá sinal alto quando um movimento for percebido, ajustando a sensibilidade é possível aumentar ou diminuir a distância percebida pelo sensor. A Figura 5 apresenta uma imagem deste sensor de movimento que foi utilizado no projeto.

O botão utilizado foi um botão com trava de 6 terminais, assim, seus contatos funcionam de forma que ao pressionar o botão é mantido um sinal normalmente aberto ou normalmente fechado, ou seja, com o botão tendo na sua entrada um sinal alto, quando o botão for pressionado o sinal de saída será sempre alto ou sempre baixo e não por pulsos. Os LEDs utilizados foram um LED vermelho para simular o alarme do sensor de presença e um LED verde para simular o acionamento da iluminação, ambos com tensão de operação de 1,9-2,1V e como a tensão de saída da GPIO da Beaglebone Black é de 3,3V foram utilizados alguns resistores para manter a segurança do LED. Para realizar as conexões entre estes componentes utilizou-se de jumpers macho-macho e macho-fêmea.

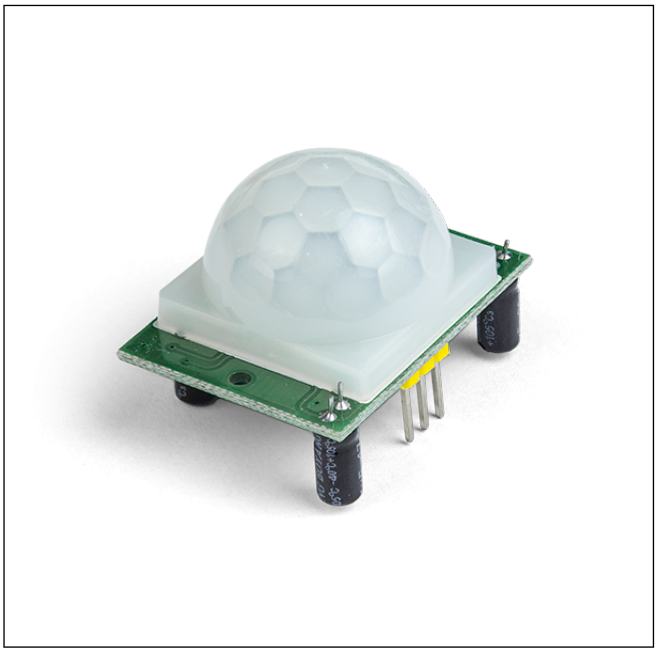

Figura 5 – Sensor de movimento e presença PIR DYP-ME003
Fonte: (Robocore, 2021)

4.1.2 Estruturação do circuito

Utilizando a Protoboard para a montagem dos circuitos, 2 pinos GPIO da placa Beaglebone Black foram utilizados como dispositivo de entrada. Em um deles a entrada corresponde ao sensor de movimento e presença, para simular um sistema de alarme, e em outro foi utilizado um botão, para simular a ativação remota de iluminação. Outros 2 pinos da GPIO foram utilizados como dispositivos de saída, em ambas as saídas foram utilizados LEDs para representar tanto o acionamento do alarme quanto a ativação da iluminação. A Figura 6 exibe o circuito montado através da ferramenta *easyeda*.

Conforme pode-se verificar, na Figura 6 o componente U1 (BEAGLEBONE OUTLINE) representa todas as entradas e saídas dos barramentos dos pinos da GPIO da Beaglebone Black (P8 e P9), permitindo a visualização de como o circuito será montado para o devido funcionamento do sistema.

Neste circuito foram definidos os pinos GPIO1_14 e GPIO2_1 como pinos de saída e os pinos GPIO0_27 e GPIO1_15 como pinos de entrada, sendo que em código foi montada a estrutura para que a GPIO1_14 seja acionada pelo sinal recebido no pino de entrada GPIO1_15 e a GPIO2_1 seja acionada pelo pino GPIO0_27. Na Figura 6 o botão é representado pela chave SW1, então o LED2 será ligado ou desligado quando o botão for pressionado. Além disso, o sensor de presença é representado pelo componente U2 (PIR Motion Sensor), portando ao receber sinal de tensão do sensor de movimento o LED1 será acionado.

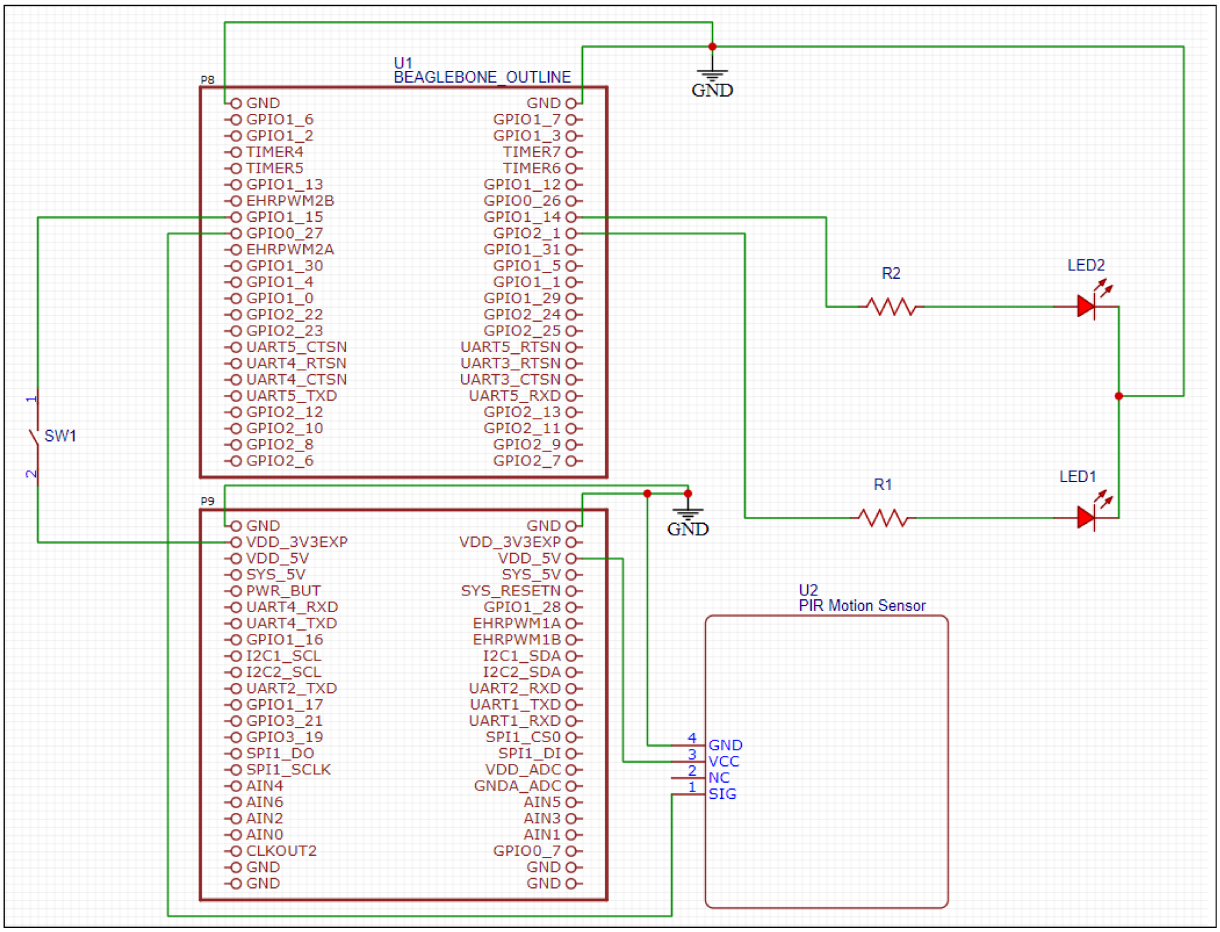

Figura 6 – Modelo do circuito eletrônico

4.2 Código da aplicação

Para o funcionamento do circuito que foi projetado, foi necessário passar por algumas etapas. Primeiramente deve-se configurar os pinos que são utilizados nos modos de funcionamento desejados, assim, os pinos GPIO1_14 (P8_16) e GPIO2_1 (P8_18) devem ser definidos como pinos de saída e os pinos GPIO0_27 (P8_17) e GPIO1_15 (P8_15) como pinos de entrada. Para isso, na função *main* utilizou-se das funções criadas na *usercall* para iniciar estas GPIOs conforme o Código 20.

Código 20 – Inicialização das GPIOs nos seus modos de operação

```
1 gpioinit(P8_15,GPIO_INPUT);
2 gpioinit(P8_16,GPIO_OUTPUT);
3 gpioinit(P8_17,GPIO_INPUT);
4 gpioinit(P8_18,GPIO_OUTPUT);
```

Para o funcionamento do projeto os pinos de entrada devem ser constantemente verificados, se houver sinal alto em algumas das entradas deve ser enviado sinal alto para os pinos

de saída correspondentes, senão deve ser enviado sinal baixo. A fim de executar essa verificação indefinidamente, foi criada a função *task1* que fica em *loop* enquanto a placa estiver ligada, essa função é iniciada dentro da função *main* como uma nova tarefa na aplicação e então é iniciado um escalonador do tipo Round-Robin (COSTA, 2014) para gerenciar seu funcionamento, conforme pode ser visto no Código 21.

Código 21 – Criação da nova tarefa para execução do processamento

```
1   int t1;
2   taskcreate(&t1,task1);
3   start(RR);
```

A *thread task1* é então iniciada e conforme pode ser verificado no Código 22 a cada período de tempo de 500 ms ela realiza a verificação. Se o pino P8_15, que recebe o sinal proveniente do botão, possui valor alto então deve ser enviado sinal alto para o pino P8_16, senão deve ser enviado sinal baixo para o pino P8_16. De forma análoga, se o pino P8_17, que recebe o sinal do sensor de presença, possui valor alto então deve ser enviado sinal alto para o pino P8_18, senão deve ser enviado sinal baixo para o pino P8_18.

Código 22 – Tarefa responsável pela leitura e escrita nas GPIOs

```
1  void task1()
2  {
3    while(1)
4    {
5      int var1 = 0;
6      gpioreadin(P8_15,&var1);
7      if (var1 == 1)
8      { gpiowrite(P8_16,HIGH);}
9      else
10     { gpiowrite(P8_16,LOW);}
11     int var2 = 0;
12     gpioreadin(P8_17,&var2);
13     if (var2 == 1)
14     { gpiowrite(P8_18,HIGH);}
15     else
16     { gpiowrite(P8_18,LOW);}
17     msleep(500);
18   }
19 }
```

Como possível perceber nestes códigos, a implementação das chamadas GPIO se tornam bem simplificadas para o programador, possuindo uma interface amigável e de fácil utilização com as funções criadas na *usercall*. O código completo da aplicação pode ser encontrado no Código 24 no Anexo B.

4.3 Montagem do circuito e resultados

Com a modelagem e o código já finalizados foi possível então montar o circuito e verificar seu funcionamento. Conforme citado anteriormente, para a montagem do circuito foi utilizada uma Protoboard, que permitiu realizar as devidas conexões como projetado. Os acionamentos e LEDs das duas simulações foram distribuídas em lados opostos da protoboard a fim de facilitar sua visualização. A Figura 7 apresenta o circuito montado com ambas as entradas ativas, sendo do lado esquerdo o acionamento da simulação de um alarme e o lado direito a simulação da ativação remota de uma iluminação.

A implementação do modelo funcionou perfeitamente, de forma que, o sensor possui uma resposta praticamente imediata a variações de movimento no ambiente, assim como, ao pressionar o botão, o LED correspondente à iluminação é ligado ou desligado, conforme o esperado.

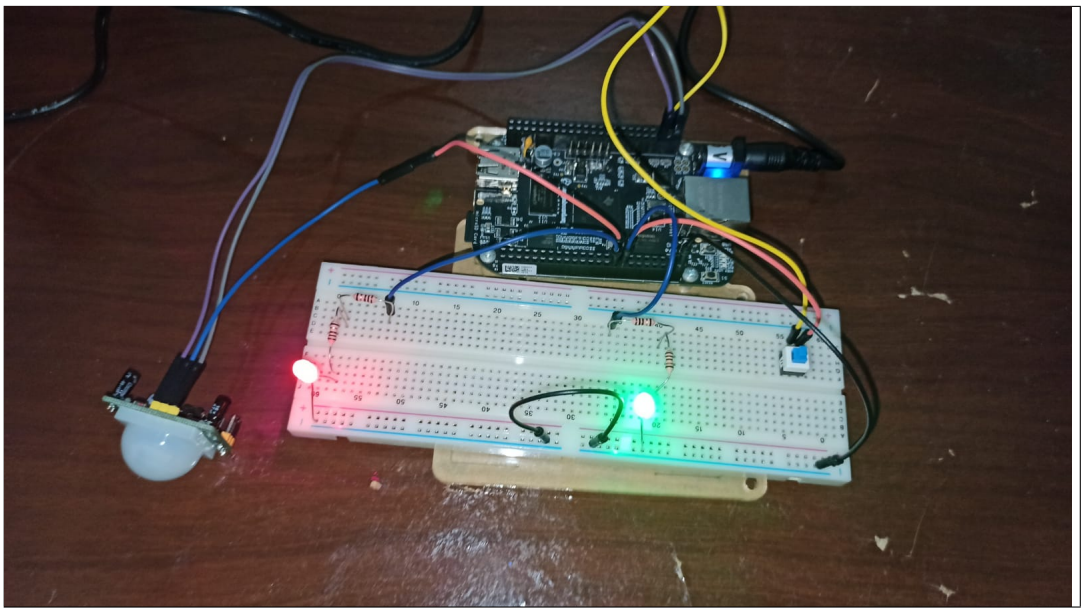

Figura 7 – Circuito montado com entradas ativas

Apesar deste ser um sistema simplificado, estes acionamentos podem ser utilizados das mais diversas formas, se forem utilizados relés por exemplo, é possível acionar sistemas de alta tensão em vez de apenas um LED. Também é possível utilizar diversos tipos de dispositivos de entrada para ativar ou desativar uma saída, aumentar a complexidade do código utilizando diversas condições para o acionamento, e com a grande quantidade de pinos disponíveis na Beaglebone Black a complexidade do sistema pode ficar muito grande com a interação de todas estas ferramentas.

5 CONCLUSÃO

Durante a elaboração desse trabalho, muitas dificuldades foram encontradas. Muito estudo foi necessário para que fosse possível entender como poderia ser implementada a comunicação com a GPIO, sendo que em sistemas operacionais como o Linux por exemplo, a programação à GPIO é feita através do acesso ao sistema de arquivos a nível de usuário e utiliza uma biblioteca própria que lida com este tipo de informação. Como o NKE ainda não possui sistemas de arquivos, outras formas de comunicação tiveram que ser implementadas, partindo para a comunicação direta com os registradores da controladora da placa.

A comunicação com os registradores também se provou uma tarefa difícil, necessitando da utilização de código assembly para interagir com estes registradores. O código foi escrito sem que fosse possível verificar onde estava ocorrendo os erros caso o código parasse dentro da execução da parte assembly, já que a verificação de erros durante a execução é feita por impressões do código executado, fato que ocorreu diversas vezes durante o desenvolvimento.

Por fim, esses obstáculos foram ultrapassados e os resultados esperados foram alcançados. Os objetivos deste trabalho foram cumpridos, criando todo o fluxo de comunicação com a GPIO da Beaglebone Black e elaborando um estudo de caso onde estas funcionalidades funcionam plenamente.

Com este desenvolvimento foi possível perceber que a programação em nível de *kernel* não é nada trivial, sendo necessário considerar os mais diversos quesitos para a implementação de uma nova funcionalidade como a que foi descrita neste trabalho. Durante a implementação da comunicação com a GPIO foi preciso aprofundar os conhecimentos, principalmente de programação assembly, linguagem que até então havia sido pouco estudada. Apesar das dificuldades, a implementação da comunicação com a GPIO utilizando o NKE abre várias possibilidades de utilização para este sistema operacional. Com a interação entre a *kernel* e diversos componentes eletrônicos através da GPIO, o NKE pode ser utilizado de uma forma mais abrangente tanto no campo educacional quanto de forma prática, sendo possível utilizá-lo no funcionamento de sistemas embarcados reais.

Durante o desenvolvimento do projeto foram verificadas diversas melhorias que podem ser feitas no NKE para a Beaglebone Black, além de novas funcionalidades que podem ser implementadas para aumentar as possibilidades de utilização do NKE. Como por exemplo a comunicação PWM (*Pulse Width Modulation*), que permitiria a utilização das GPIOs de forma ainda mais abrangente, podendo acionar servomotores, *buzzers* etc, além também de aumentar o número de sensores que poderiam ser utilizados, como por exemplo um sensor de temperatura, que poderia ser utilizado de formas diferentes tanto para leitura direta da temperatura pelo terminal quanto para uso automatizado acionando algum sistema quando atinge certas condições.

REFERÊNCIAS

ARM KEIL. **ARM Assembler User Guide**. Disponível em: <https://www.keil.com/support/man/docs/armasm/>. Acesso em: 6 mai. 2021.

BEAGLEBOARD.ORG. **BeagleBone 101**. Disponível em: <https://beagleboard.org/Support/bone101>. Acesso em: 27 mai. 2021.

BRAND, Jamile Santis. **Automatizações Do Teste Dos Algoritmos De Escalonamento No Nanokernel NKE**. 2016. TCC – Universidade Estadual do Rio Grande do Sul, Guaíba.

BURJAILI, Ivan; OLIVEIRA, Frederico Fernandes de. **Manual de Acesso a Placa BeagleBone Black Usando o Sistema Operacional NKE**. 2013.

COSTA, C. M. da et al. **NKE - Um Nanokernel Educacional para Microprocessadores ARM**. IV Simpósio Brasileiro de Engenharia de Sistemas Computacionais, Manaus, nov. 2014.

CRUZ, Ariadne Arrais; LISBOA, Emerson Fausto. WebHome – Automação residencial utilizando Raspberry PI. **Revista Ciência e Tecnologia**, 2014. Disponível em: <http://www.revista.unisal.br/sj/index.php/123/article/view/365>. Acesso em: 29 jun. 2021.

CUNHA, Alessandro. O que são sistemas embarcados? **Revista Saber Eletrônica**, Editora: Saber, São Paulo, 2007.

EDWARDS, S. A. et al. **Design of Embedded Systems: Formal Models, Validation, and Synthesis**. PROCEEDINGS OF THE IEE, California University, Berkeley, v. 85, n. 3, p. 366–390, mar. 1997.

FILIPEFLOP. **Protoboard 830 Pontos**. Disponível em: <https://www.filipeflop.com/produto/protoboard-830-pontos/>. Acesso em: 21 fev. 2021.

FREITAS, S. A. G. de. **Desenvolvimento De Um Sistema De Controle Embarcado Para A Equipe De Futebol De Robôs Araranguá Intruders**. Fev. 2013. TCC – Universidade Federal de Santa Catarina, Araranguá.

FURBER, S. B. **ARM System-On-Chip Architecture**. Addison-Wesley, 2000.

HERDER, N. **Toward a True Microkernel Operating System**. Fev. 2005. Master of Science thesis in Computer Science – Vrije Universiteit, Amsterdam.

MAZIERO, Carlos A. **Sistemas Operacionais:Conceitos e Mecanismos**. 2019.

MURLIKY, Lucas. **Concepção E Implementação De Um Nanokernel Para Sistemas Embarcados E De Tempo Real**. 2014. TCC – Universidade Estadual do Rio Grande do Sul, Guaíba.

OSDEV.ORG. **Kernels**. Disponível em: <https://wiki.osdev.org/Kernels>. Acesso em: 27 jun. 2021.

ROBOCORE. **Sensor de Presença PIR - HC-SR501**. Disponível em: <https://www.robocore.net/sensor-ambiente/sensor-de-presenca-pir-hc-sr501>. Acesso em: 15 mar. 2021.

SEVERO, Renato. **Implementação de um Algoritmo de Escalonamento Particionado para arquiteturas Multi-core no RT-NKE**. 2016. Monografia – Universidade Estadual do Rio Grande do Sul.

SILVA, Leonardo Da Luz. **Concepção e implementação de um modelo de programação para um nanokernel paralelo.** 2014. TCC – Universidade Estadual do Rio Grande do Sul, Guaíba.

SUNXI.ORG. **UEnv.txt**. Disponível em: <https://linux-sunxi.org/UEnv.txt>. Acesso em: 22 mar. 2021.

TEXAS INSTRUMENTS. **AM335x Sitara™ Processors Datasheet**. 2011. Disponível em: <https://www.ti.com/lit/gpn/am3358>. Acesso em: 10 abr. 2021.

TEXAS INSTRUMENTS. **AM335x Sitara™ Processors Technical Reference Manual**. 2011. Disponível em: <https://www.ti.com/lit/pdf/spruh73>. Acesso em: 12 mar. 2021.

ANEXO A – DETALHES DO CÓDIGO ASSEMBLY

Código 23 – Código completo da gpiohandler.S

```
1  .text
2  .data
3
4  .global InitGpioOutput, InitGpioInput, SetGpioOutOn,
       SetGpioOutOff, GetGpioOutValue, GetGpioInValue
5
6  //*************************************************************
7  //*                    Function InitGpioOutput
8  //*************************************************************
9  //Funcao que seta o pino especificado como output
10 //
11 InitGpioOutput:
12         stmfd sp!, {r2-r8, lr}
13         .equ GPIO_OE, 0x134
14         MOV          r3, #GPIO_OE
15         ADD          r7,r0,r3
16         SWP          r7,r7,[r8]
17         LDR          r7,[r8]
18
19         ORR          r5,r7,r1
20         SUB          r4,r5,r1
21         STR          r4,[r7]
22
23         ldmfd sp!, {r2-r8,lr}
24         MOV          pc,lr
25
26 //*************************************************************
27 //*                    Function InitGpioInput
28 //*************************************************************
29 //Funcao que seta o pino especificado como input - no reset os
       pinos ficam em input
30 //
31 InitGpioInput:
32         stmfd sp!, {r2-r8, lr}
33         .equ GPIO_OE, 0x134
34
35         MOV          r3, #GPIO_OE
```

```
36          ADD         r7,r0,r3
37          SWP         r7,r7,[r8]
38          LDR         r7,[r8]
39
40          ORR         r1,r7,r1
41          STR         r1,[r7]
42
43          ldmfd sp!, {r2-r8,lr}
44          MOV         pc,lr
45
46  //*************************************************************
47  //*                   Function SetGpioOutOn
48  //*************************************************************
49  //Funcao que seta valor do pino que esta como output com valor
        alto(1)
50  //
51  SetGpioOutOn:
52          stmfd sp!, {r2-r8, lr}
53          .equ GPIO_SETDATAOUT, 0x194
54
55          MOV         r3, #GPIO_SETDATAOUT
56          ADD         r7,r0,r3
57          SWP         r7,r7,[r8]
58          LDR         r7,[r8]
59
60          ORR         r1,r7,r1
61          STR         r1,[r7]
62
63          ldmfd sp!, {r2-r8,lr}
64          MOV         pc,lr
65
66  //*************************************************************
67  //*                   Function SetGpioOutOff
68  //*************************************************************
69  //Funcao que seta valor do pino que esta como output com valor
        baixo(1)
70  //
71  SetGpioOutOff:
72          stmfd sp!, {r2-r8, lr}
73          .equ GPIO_CLEARDATAOUT, 0x190
74          MOV         r3, #GPIO_CLEARDATAOUT
75          ADD         r7,r0,r3
```

```
76          SWP          r7,r7,[r8]
77          LDR          r7,[r8]
78
79          ORR          r1,r7,r1
80          STR          r1,[r7]
81
82          ldmfd sp!, {r2-r8,lr}
83          MOV          pc,lr
84
85  //*************************************************************
86  //*                    Function GetGpioOutValue
87  //*************************************************************
88  //Funcao que retorna valor do pino que esta como output
89  //
90  GetGpioOutValue:
91          stmfd sp!, {r2-r8, lr}
92          .equ GPIO_DATAOUT, 0x13c
93
94          LDR          r0,[r0,#GPIO_DATAOUT]
95          AND          r0,r0,r1
96
97          ldmfd sp!, {r2-r8,lr}
98          MOV          pc,lr
99
100 //*************************************************************
101 //*                    Function GetGpioInValue
102 //*************************************************************
103 //Funcao que retorna valor que chega no pino que esta como input
104 //
105 GetGpioInValue:
106         stmfd sp!, {r2-r8, lr}
107         .equ GPIO_DATAIN, 0x138
108
109         LDR          r0,[r0,#GPIO_DATAIN]
110         AND          r0,r0,r1
111
112         ldmfd sp!, {r2-r8,lr}
113         MOV          pc,lr
```

ANEXO B – DETALHES DA APLICAÇÃO MAIN

Código 24 – Código completo da main.c

```
#include "kernel/kernel.h"
#include "kernel/usercall.h"
#include "kernel/scheduler.h"
#include "board/uart.h"
#include "board/gpio.h"

void task1()
{
  while(1)
  {
    int var1 = 0;
    gpioreadin(P8_15,&var1);
    if (var1 == 1)
    { gpiowrite(P8_16,HIGH);}
    else
    { gpiowrite(P8_16,LOW);}
    int var2 = 0;
    gpioreadin(P8_17,&var2);
    if (var2 == 1)
    { gpiowrite(P8_18,HIGH);}
    else
    { gpiowrite(P8_18,LOW);}
    msleep(500);
  }
}

int main(void)
{
  gpioinit(P8_15,GPIO_INPUT);
  gpioinit(P8_16,GPIO_OUTPUT);
  gpioinit(P8_17,GPIO_INPUT);
  gpioinit(P8_18,GPIO_OUTPUT);
  int t1;
  taskcreate(&t1,task1);
  start(RR);
  return 0;
}
```