UNIVERSIDADE FEEVALE

EVERTON JERFERSON DA SILVA

DESENVOLVIMENTO DE SOFTWARE EMBARCADO UTILIZANDO A MODELAGEM DE AGENTES INTELIGENTES

Novo Hamburgo

2016

EVERTON JEFERSON DA SILVA

DESENVOLVIMENTO DE SOFTWARE EMBARCADO UTILIZANDO A MODELAGEM DE AGENTES INTELIGENTES

Trabalho de Conclusão de Curso
apresentado como requisito parcial
à obtenção do grau de Bacharel em
Ciência da Computação pela
Universidade Feevale

Orientador: Ricardo Ferreira de Oliveira

Novo Hamburgo
2016

AGRADECIMENTOS

Gostaria de agradecer a todos os que, de alguma maneira, contribuíram para a realização desse trabalho de conclusão, em especial: a minha mãe e toda minha família.

Aos amigos e às pessoas que convivem comigo diariamente, minha gratidão, pelo apoio emocional - nos períodos mais difíceis do trabalho.

RESUMO

Desde o surgimento da chamada ‘Internet das coisas’, muitas expectativas vem sendo criadas sobre seu futuro, bem como diversos estudos sobre o tema. Com isso diversos desafios também surgiram, principalmente devido aos problemas ocasionados da integração de um *hardware* com um *software* capaz de gerir o equipamento de forma inteligente. Uma solução para esse tipo de desenvolvimento pode ser a utilização da modelagem baseada em agentes inteligentes, um tema muito discutido e estudado na área da inteligência artificial. Diversos estudos demostram essa modelagem sendo mais recomendável para ambientes complexos. Porém deve-se ressaltar a complexidade que aumenta no desenvolvimento de projetos utilizando essa abordagem. Tendo em visto essa problemática, esse trabalho se propõe em estudar e apresentar um *framework* para o desenvolvimento de sistema embarcado utilizando agentes inteligentes. Esse *framework* deve trazer diversos benefícios para esse tipo de desenvolvimento, como padronizações, abstração de funcionalidades básicas, divisão de responsabilidades entre outras.

Palavras-chave: Internet das Coisas. Sistemas embarcados. Agentes Inteligentes. *Framework*.

ABSTRACT

Since the emergence of so-called 'Internet of Things', many expectations are being raised about its future, as well as several studies on the topic. In accordance with this, many challenges have also emerged, mainly due to problems caused the integration of hardware with software capable of managing intelligently equipment. A solution to this type of development may be the use of intelligent agent based modeling, a hotly debated topic and studied in the field of artificial intelligence. Several studies demonstrate that modeling is more recommended for complex environments. But it should be noted that increases the complexity in the development of projects using this approach. Having seen this problem, this paper aims to study and present a framework for the development of embedded system using intelligent agents. This framework should bring many benefits to this type of development, such as standardization, abstraction basic functionality, division of responsibilities among others.

Keywords: Internet of Things. Embedded systems. Intelligent Agents. Framework.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

A/D	*Analog-To-Digital*
ACG	*Apollo Guidance Computer*
AID	*Agenteidentifier*
AOP	*Agent-Oriented Programming*
AUML	*Agent Unified Modeling Language*
BDI	*Belief-Desire-Intention*
CI	Circuíto integrado
CPU	*Central Processing Unit*
D/A	*Digital-To-Analog*
EEPROM	*Electrically Erasable Programmable Read Only Memorya*
EPROM	*Electrically Programmable Only Memory*
FIPA	*Foundation For Intelligent Physical Agents*
HOL	*High-order languagens*
I/O	*Input/Output*
IDE	*Integrated development environment*
IEEE	*Institute Of Electric And Electronic Engineers*
JADE	*Java Agent Development Framework*
LCD	*Liquid Crystal Display*
LED	*Light Emitting Diode*
OO	Orientação a Objeto
OSI	*Open Systems Interconnection*
PC	*Personal Computer*
PROM	*Programmable Read Only Memory*
PRS	*Procedural Reasoning System*
RAM	*Random Acess Memory*
ROM	*Read Only Memory*
RTOS	*Real Time Operating System*
SD	*Secure Digital*
SMA	*Sistemas Multi-Agentes*
UML	*Unified Modeling Language*

VHLL *Very-high-level languagens*

SUMÁRIO

INTRODUÇÃO

Um tema que é muito discutido e estudado na atualidade é a chamada ‘Internet das coisas’, também chamada de IoT (*Internet of Things*). O termo ‘Internet das coisas’ pode ser entendido como objetos, que através de endereçamentos únicos, são capazes de interagir uma com a outra e cooperar com seus vizinhos para alcançar objetivos comuns (Atzori, Lera e Morabito, 2010).

O termo interação, no contexto de comunicação, é o ato de comunicar-se ou realizar ações com outros indivíduos do mesmo grupo. Assim é necessário que os objetos, citados acima, sejam de fato componentes de *hardware* equipados com sensores e atuadores. Outro ponto, que deve ser apresentado, é o controle desse objeto, pois para que a interação aconteça entre um objeto e outro, é necessário que cada um seja independente. Por tanto, esses objetos devem tratar-se de equipamentos controlados por um sistema computacional, caracterizando-os como um sistema embarcado.

Um sistema embarcado, ou por vezes chamado de firmware, pode ser definido como um dispositivo que contém componentes fortemente acoplado de *hardware* e *software* para executar uma função. Faz parte de um sistema maior, não se destina a ser, independentemente, programável pelo utilizador, e geralmente trabalha com pouca ou nenhuma interação humana. (Jiménez, Palomera, e Couvertier, 2014). O desenvolvimento desse tipo sistema, possui diversas dificuldades, pois diferentemente de um sistema computacional comum, possuem métricas de eficiência especificas (Marwedel, 2006), como:

- Consumo de Energia: a maioria dos sistemas embarcados são móveis e por isso utilizam baterias portanto, a eficiência energética sempre deverá ser considerada.
- Tamanho de código: todo o código do sistema estará embarcado junto, e geralmente numa mesma memória, ou seja, quanto menor o código mais memória livre para execução do sistema.
- Execução eficiente: o sistema deverá apenas conter componentes realmente necessários para a execução das tarefas propostas, isso visando um menor consumo energético e maior simplicidade do sistema.

- Peso: sistemas embarcados geralmente são móveis, sendo assim devem ser leves, facilitando a locomoção com baixa custo energético.
- Custo: a boa utilização de componentes de *hardware* no sistema pode baixar muito o custo, e um sistema com baixo custo tende a ser mais atrativo para comercialização.

Além das dificuldades de desenvolvimentos resultantes do *hardware*, também é importante ressaltar a dificuldade que é gerada no desenvolvimento de um sistema inteligente. Pois, como citado, o sistema deve ser autônomo para conseguir interagir um com o outro, de maneira que se faz necessário que ele possua um certo grau de inteligência. Uma solução para o desenvolvimento de um sistema inteligente é o uso da modelagem baseada em agentes. Sendo o uso de modelagem de agentes uma abordagem mais recomendável que a modelagem de *software* convencional, onde há a necessidade da realização de tarefas que precisam de alto desempenho e/ou onde o ambiente é considerado complexo (Rao e Georgeff, 1995).

Apesar de não existir um consenso sobre a definição de agentes, no campo da Inteligência artificial, se pode citar um agente como sendo um sistema computacional que está situado em algum ambiente e que é capaz de ações autônomas nesse ambiente em ordem de satisfazer seus objetivos (Russel e Norvig, 2003). Basicamente, é possível entender um agente inteligente como sendo uma entidade capaz de perceber seu ambiente e capaz de interagir de forma a realizar seus objetivos. Algumas outras características também são citadas por Wooldridge e Jennings (1995), como:

- Autonomia: agentes operarão sem a intervenção humana ou outra, eles tem controle sobre suas ações e seus estados internos.
- Habilidade social: agentes interagem com outros agentes e possivelmente humanos também.
- Reatividade: agentes percebem seu ambiente e respondem às alterações ocorridas.
- Pro-atividade: agentes não agem simplesmente por reação ao ambiente, eles também tem o comportamento de agir para um objetivo, tomando a iniciativa.

Como visto até agora, o desenvolvimento de um sistema embarcado é complexo e utilizar a abordagem de agentes inteligentes deve torná-lo ainda mais complexo. Isto faz necessário o desenvolvimento de técnicas capazes de simplificar o desenvolvimento desse tipo de projeto. Para Nebel e Schumacher (1996) as chaves para gerenciar a complexidade do

projeto e o aumento do esforço de designer são a hierarquia e a abstração. Pois a decomposição hierárquica estrutural do problema de projeto reduz a uma complexidade por subproblemas de isolamento e, portanto, faz o problema de *designer* acessível para uma solução. Carro e Wagner (2003) também defendem a mesma ideia:

> Devido à possível complexidade da arquitetura de um sistema embarcado, contendo múltiplos componentes de *hardware* e *software* em torno de uma estrutura de comunicação, e à grande variedade de soluções possíveis visando o atendimento de requisitos de projeto, como desempenho, consumo de potência e área ocupada, é essencial elaborar o projeto do sistema em níveis de abstração elevados e utilizando ferramentas que automatizem ao máximo as diversas etapas de uma metodologia consistente com os desafios existentes (CARRO e WAGNER, 2003, p.15).

Sendo assim, uma solução para simplificar o desenvolvimento de sistemas embarcados inteligentes, seria o desenvolvimento de um framework específico para projetos desse tipo. Por isso este trabalho tem como objetivo propor e desenvolver um *framework* destinado ao desenvolvimento de sistemas embarcados inteligentes. Um *framework*, como o nome sugere é um estrutura genérica estendida para criar uma aplicação ou subsistema específico, que fornecem suporte para recursos genéricos, suscetíveis de serem usados em todas as aplicações de tipos semelhantes (Sommerville, 2011).

A arquitetura do framework proposto deverá seguir o conceitos de decomposição hierárquica, visando a abstração dos seus componentes. Inicialmente, a arquitetura deverá ser constituída de pelo menos 4 camadas, sendo elas, uma camada para isolamento do *hardware*, uma para funcionamento básico do *hardware*, uma para o interpretador e uma camada contendo a aplicação. Essa implementação trará uma série de vantagens para o desenvolvimento de projetos de sistemas embarcados inteligentes, como: padronização de um modelo de desenvolvimento, trazendo mais agilidade ao processo; encapsulamento das funcionalidades mais básicas diretamente no framework, evitando assim retrabalho em outros projetos; abstração do funcionamento básico de *hardware*, não havendo assim a necessidade de conhecer códigos de mais baixo nível.

No capítulo 1 abordar-se-á os sistemas embarcados, suas características técnicas e softwares relacionados. No capítulo 2 será apresentada a técnica de agentes e seus ambientes, suas arquiteturas destacando-se os sistemas multi-agentes. No capítulo 3 se discorrerá sobre os frameworks para sistemas embarcados baseados em agentes, juntamente com a proposta de framework do presente trabalho, descrevendo sua implementação no capítulo 4. Por fim, no capítulo 5 será apresentado o projeto piloto, sua especificação e desenvolvimento. Serão apresentados os procedimentos de validação e análise, concluindo-se no capítulo 6.

1 SISTEMAS EMBARCADOS

Os sistemas embarcados surgiram na década de 60, proporcionados pela corrida espacial. O primeiro sistema considerado como embarcado, na concepção atual, foi o chamado Apollo Guidance Computer (AGC), que consistia no computador de bordo utilizado no projeto Apolo, um sistema com apenas 4KB de RAM, que controlava a navegação da espaçonave (Jiménez, Palomera, e Couvertier, 2014).

Um sistema embarcado é definido por Andrade e Oliveira (2006) como ‘sistemas que possuem uma capacidade de processamento de informações vinda de um *software* que está sendo processado internamente nessa unidade. Ou seja, o *software* está embarcado na unidade de processamento.’ Assim pode ser definido, como um *software* específico projetado para um determinado *hardware*, cujo *software* é embutido diretamente no *hardware*, sendo que geralmente, tanto *hardware* quanto *software*, são projetados para realizar, em conjunto, tarefas específicas. As principais características são descritas por Peter Marwedel (2006) como sendo:

- Sistemas embarcados quase sempre são projetados para interagir com o ambiente externo, por meio de sensores que coletam informações e atuadores para controlar o ambiente.
- Sistemas embarcados devem ser confiáveis: a maioria desses sistemas realizam tarefas críticas e não tolerantes a falhas, pois uma falha poderá causar um resultado danoso ao seu usuário. Sendo assim, para um sistema ser considerado confiável deve ter as seguintes características:
    - Estabilidade: é a probabilidade que um sistema não irá falhar.
    - Manutenção: é a probabilidade que uma falha no sistema será corrigida em um certo intervalo de tempo.
    - Disponibilidade: é a probabilidade de que um sistema estará disponível em certo tempo. Alta estabilidade e recuperação levam a uma alta disponibilidade.
    - Segurança: um sistema deve ser seguro tanto nos dados que possui e nos dados que utiliza para comunicação.
- Sistemas embarcados possuem outras métricas de eficiência, diferentes dos sistemas convencionais. Abaixo estão algumas delas:

- Consumo de Energia: a maioria dos sistemas embarcados são móveis e por isso utilizam baterias, portanto a eficiência energética sempre deverá ser considerada.
- Tamanho de código: todo o código do sistema estará embarcado junto, e geralmente numa mesma memória, ou seja, quanto menor o código mais memória livre para execução do sistema.
- Execução eficiente: o sistema deverá apenas conter componentes realmente necessários para a execução das tarefas propostas, isso visando um menor consumo energético e maior simplicidade do sistema.
- Peso: sistemas embarcados geralmente são móveis, sendo assim devem ser leves, facilitando a locomoção com baixa custo energético.
- Custo: a boa utilização de componentes de *hardware* no sistema pode baixar muito o custo, e um sistema com baixo custo tende a ser mais atrativo para comercialização.

- Sistemas embarcados são desenvolvidos para realizar uma função ou um conjunto de funções específicas e nunca para serem programados pelo usuário. Ele pode mudar configuração do equipamento, mas não poderá acrescer funções novas.
- Grande parte dos sistemas embarcados não possui dispositivos de interface típicos de outros sistemas, como teclado, monitor e mouse, e sim utilizam dispositivos como botões, visor LCD ou LEDs. Isso faz com que o usuário dificilmente consiga ver o que realmente está sendo processado no sistema.
- Em alguns casos os sistemas embarcados devem operar com requisitos de sistemas de tempo real, assim, caso algum cálculo não seja executado no tempo esperado, pode ser considerado uma falha no sistema, sendo que isso sempre deve ser evitado.
- A maioria dos sistemas embarcados é reativa ao ambiente, sendo assim quase sempre estão esperando algum sinal externo para executar uma tarefa.

Tammy Noergaard (2012) defende uma arquitetura de um sistema embarcado dividindo ela em 3 camadas, conforme mostra a Figura 1, sendo elas, uma camada de *hardware* e duas camadas de *software*, uma contendo um sistema operacional e outra contendo uma camada de aplicação. O autor também defende que a camada de *hardware* é a única comum a todo projeto, pois pode haver apenas uma camada de *software*. Será visto nos

próximos capítulos as principais características da camada de *hardware* e das camadas de *software*.

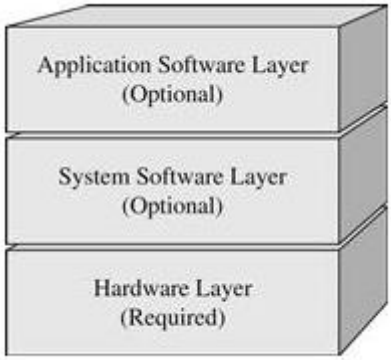

Figura 1 - Estrutura de um sistema embarcado
Fonte: Noergaard(2012)

1.1 Hardware

Os componentes de *hardware* de um sistema embarcado, consistem em componentes eletrônicos, necessários para a execução das tarefas, cujo equipamento foi projetado. Sendo assim cada projeto deve possuir um conjunto de componentes distintos entre si. Contudo, é possível determinar 3 componentes básicos e comuns a quase todo os projetos, sendo eles CPU (Central Processing Unit), memórias e portas de entrada/saída de sinais (Jiménez, Palomera, e Couvertier, 2014). A Figura 2, demostra a estrutura comum de um sistema embarcado e a interligação entre cada componente. Nos próximos sub capítulos, serão explicadas as características dos principais componentes de *hardware*.

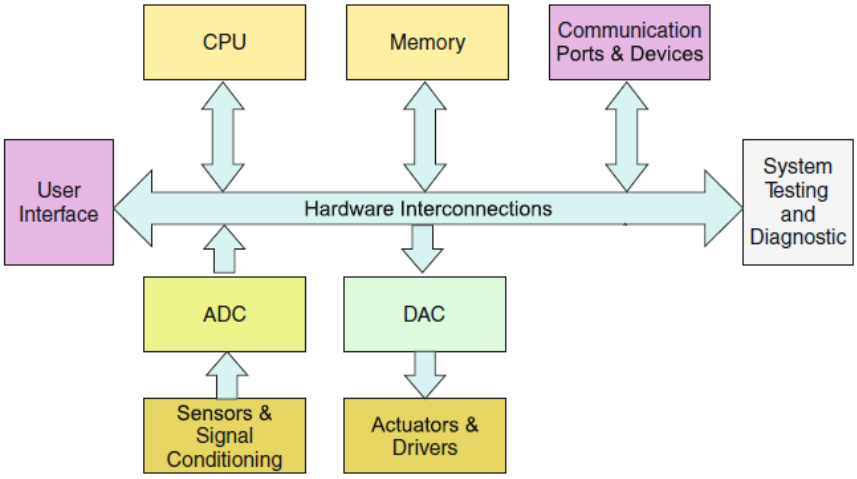

Figura 2- Comunicação entre componentes de *hardware* em sistemas embarcados
Fonte: Jiménez, Palomera e Couvertier (2014)

1.1.1 CPU

A CPU executa as instruções de *software* para processar as entradas do sistema e tomar as decisões que orientam o funcionamento do sistema (Jiménez, Palomera, e Couvertier, 2014). Geralmente as CPUs utilizadas em projetos de sistemas embarcados são diferentes dos processadores utilizados em PCs (Personal Computer) comuns, isso se deve principalmente, como visto anteriormente, ao fato que as métricas de eficiência serem diferentes em sistemas embarcados (Marwedel, 2006).

1.1.2 Memórias

As memórias em sistemas embarcados além de armazenar o *software* também são necessárias para armazenar dados temporários. Sendo as principais características das memórias eletrônicas (Andrade e Oliveira (2006)):

- Tempo de acesso: é o tempo necessário para acessar a memória, para ler ou gravar dados.
- Capacidade: quantidade efetiva para armazenamento de dados.
- Não-Volatilidade: capacidade de a memória manter os dados sem alimentação elétrica.
- Tempo de latência: é o intervalo mínimo entre operações de leitura e escrita.

Levando em considerações essas características, Andrade e Oliveira (2006) também citam as formas de fabricação das memórias, sendo elas:

- RAM (*Random Acess Memory*): é uma memória do tipo volátil e o acesso aos dados pode ser feito de forma randômica, ou seja não é necessário percorrer todos os dados para acessar um determinado dado.
- ROM (*Read Only Memory*): são memórias do tipo não-volátil, porém não podem ser escritas e sim apenas lidas, essas memórias são geralmente gravadas pelo seu fabricante.
- PROM (*Programmable Read Only Memory*): memórias desse tipo são memórias do tipo ROM, porém são fabricadas sem nenhuma gravação, assim o projetista pode realizar sua gravação, porém essa gravação pode ser feita apenas uma vez.

- EPROM (*Electrically Programmable Only Memory*): também consiste em um tipo de memória PROM, contudo seu conteúdo pode ser apagado utilizando algum processo especial, como luz ultravioleta por exemplo. O número de vezes que a memória pode ser gravada e apagada é consideravelmente baixo.
- EEPROM (Electrically Erasable Programmable Read Only Memorya): são memórias ROM que podem ser apagadas por processos elétricos, elas podem ser apagadas e gravadas diversas vezes, contudo a gravação tende a ser bem mais lenta que a leitura.
- FLASH: essas são as memórias que geralmente são utilizadas para armazenar o *software* de um sistema embarcado. São memórias capazes de serem apagadas eletricamente, sendo que os tempo de leituras e gravação são bem superiores aos tempos de uma memória EEPROM.

1.1.3 Entradas/saídas

Os dispositivos de entrada e saída, ou também conhecidos como dispositivos de I/O (*input/output*), são os dispositivos responsáveis pela interação do sistema como o ambiente físico. Basicamente se pode definir os dispositivos de entrada como sendo os sensores e os de saídas, como sendo os atuadores. Os sensores tem como objetivo transformar grandezas físicas como temperatura, peso, velocidade entre outras, em sinais elétricos que a CPU seja capaz de interpretar. É sua função a coleta de dados. Os atuadores, são os dispositivos que geram alguma ação no ambiente físico, seja essa ação realizada com dispositivos eletromecânicos, que são capaz de realmente mudar o ambiente, ou simplesmente um display mostrando alguma informação ao usuário. Devido a um sistema embarcado ser uma espécie de computador digital, ele está limitado a trabalhar com sinais discretos, sendo assim é necessário que sinais analógicos sejam transformados em sinais digitais (Marwedel, 2006)), para isso são utilizados os dispositivos conhecidos como A/D (*analog-to-digital*) ou D/A (*digital-to-analog*), um é responsável por transformar sinais analógicos em sinais digitais e o outro digitais em analógicos respectivamente.

1.1.4 Microcontroladores

Um dos componentes mais utilizados na atualidade em sistema embarcados modernos e que merece uma atenção especial são os chamados microcontrolados, que se tratam de diversos componentes integrados em um único *chip*. Um microcontrolador possui

geralmente como componente principal um microprocessador, alinhado com memórias, de ambos os tipos (programas e dados) e dispositivos de I/O (*input/output*). Pode também possuir outros componentes, como temporizadores, conversores de dados entre outros (Jiménez, Palomera, e Couvertier, 2014). A Figura 3, mostra a estrutura interna de um microcontrolador, especificamente a do AT328p da empresa ATMEL.(DataSheet 328p). Já a Figura 4 mostra a visão física desse componente.

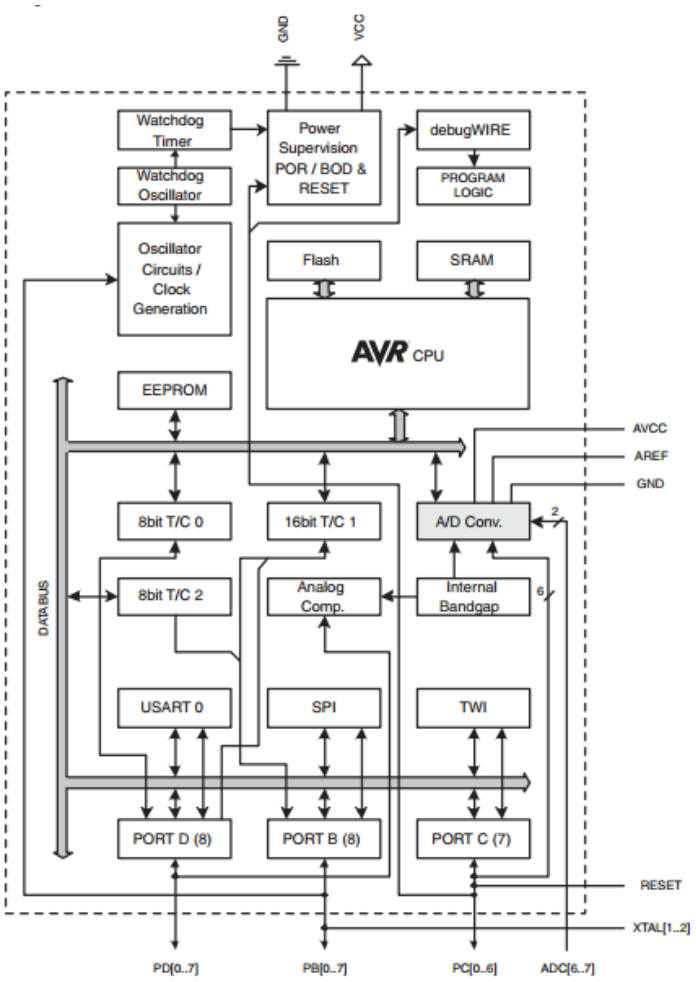

Figura 3 - Estrutura interna do AT328p
Fonte: Datasheet atmel 328p.

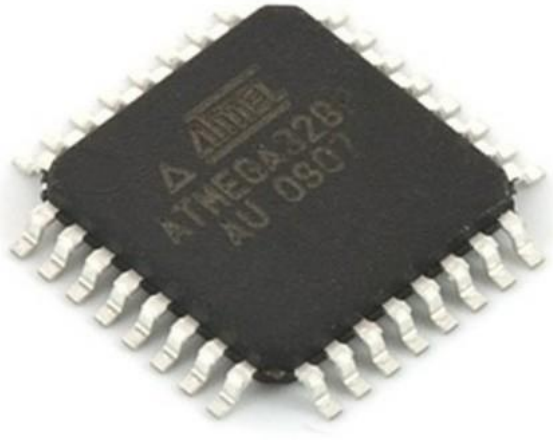

Figura 4 - Visão externa do AT328p
Fonte: http://www.institutodigital.com.br/pd-f462c-microcontrolador-atmega328p-au-smd.html

Os microntroladores podem ser ditos como microprocessadores melhorados (microprocessador apenas possui a CPU). Os microprocessadores foram um grande avanço na eletrônica, ocorrido na década de 70, porém os circuitos começavam a ficar mais complexos, pois o microprocessador necessitava de outros componentes para ser funcional. Contudo, na década de 90, com as melhorias no processamento do silício e na fabricação de CI (Circuito integrado), se conseguiu integrar outros componentes, juntamente com o microprocessador, em um mesmo *chip*, surgindo assim os chamados microcontroladores. (Trevennor, 2012). Após o seu surgimento, esse tipo de CI vem dominando os projetos eletrônicos, principalmente devido ao seu uso gerar simplificações aos projetos eletrônicos.

1.2 Software

Os componentes de *software* de sistemas embarcados, também chamados de firmware, são todos os programas necessários para a execução do *hardware* do sistema, sendo que geralmente esses programas são armazenados em algum tipo de memória não volátil. Outra característica, é que esses programas não são destinados a receber modificações em seu código pelo usuário, contudo em alguns casos eles possam receber algum tipo de atualização. (Jiménez, Palomera, e Couvertier, 2014).

O *software* embarcado, igualmente a um *software* convencional, se trata de um conjunto de linhas de código, codificadas em alguma linguagem que o processador consiga

interpretar. Andrade e Oliveira (2006) citam dois tipos de linguagens de programação, as chamadas linguagens de baixo nível e linguagens de alto nível. Uma linguagem é dita de baixo nível, quando ela é uma linguagem efetiva do *hardware*, não sendo assim necessário converte-la para que o processador possa conseguir interpretar. Uma linguagem de alto nível é caracterizada, quando existem estruturas de controles abstraídas em camadas mais inferiores do *software*, nesses casos é necessário a utilização de um compilador, que é responsável por transcrever o código em uma linguagem de baixo nível, que o processador consiga interpretar. No quadro 1 (NOERGAARD, 2012) é possível ver um breve resumo da evolução das linguagens de programação.

Quadro 1 - Evolução das linguagens de programação
Fonte: NOERGAARD, 2012

	Linguagem	Detalhes
Primeira geração	Código de máquina	Binário(0,1) e dependente de *hardware*
Segunda geração	Linguagem Assembly	Dependente de *hardware*, representando uma correspondência binária com o código de máquina.
Terceira geração	HOL (*high-order languages*) Linguagens procedurais.	Linguagens de alto nível com frases mais parecidas com o Inglês e muito mais transportáveis, tais como C e Pascal
Quarta geração	VHLL (*very-high-level languages*), Linguagens não procedurais.	Linguagens de ‘muito’ alto nível, linguagens orientadas a objetos (C ++, Java, etc.), linguagens de consulta de banco de dados (SQL), etc.
Quinta geração	Linguagens naturais	Programação semelhante à conversação, normalmente

		usada em inteligência artificial (AI). Ainda nas fases de pesquisa e desenvolvimento na maioria dos casos. Ainda não aplicáveis em sistemas embarcados.

Carro e Wagner(2003) ressaltam a importância do compilador no desenvolvimento de um sistema embarcado, principalmente devido à complexidade crescente dos projetos, sendo assim necessárias abstrações cada vez maiores para diminuir o tempo de projeto. Contudo, os autores, também defendem a utilização de linguagens de baixo níveis (Assembly, por exemplo), afim de reduzir o consumo de recursos e gerar maior desempenho no sistema.

Na maioria dos sistemas embarcados é necessário que o *software* seja, ou possua características, de um sistema de tempo real, isso porque o tempo de resposta de um sistema embarcado quase sempre é fundamental, como afirma Sommerville (2011):

> A capacidade de resposta em tempo real é a diferença crítica entre sistemas embutidos e outros sistemas de *software*, como os sistemas de informação, os sistemas baseados em Web ou os sistemas de *software* pessoal, cuja a principal finalidade é o processamento de dados.(p375)

Um sistema é dito como de tempo real sempre que o funcionamento se dá não apenas pela execução correta das tarefas, mas também por um tempo ideal de processamento, isso não quer dizer que que todo o processo ocorrerá de forma instantânea, mas sim dentro do tempo máximo estipulado como objetivo do sistema (Andrade e Oliveira, 2006)

1.2.1 RTOS

Um sistema operacional de tempo real ou *real time operating system* (RTOS) é um sistema operacional cujo funcionamento, além de cumprir as características de um sistema operacional convencional, também depende do atendimento correto de requisitos temporais associados à execução dos processos, tais como tempos máximos de execução e dos intervalos de tempo entre inícios sucessivos de execução de um processo. (Carro e Wagner, 2003).

Os conceitos de tempo real, em um RTOS, estão principalmente ligados ao escalonador de tarefas, como afirmam Carro e Wagner:

> A principal consequência das restrições temporais incide sobre o escalonamento de tarefas, função associada à gerência de processos. Num RTOS, tarefas sempre têm uma prioridade associada, definida de acordo com critérios que podem variar, como forma de garantir o atendimento das restrições temporais, e devem ser preemptivas, ou seja, devem poder ser interrompidas por tarefas de maior prioridade (Carro e Wagner, 2003).

Existem algumas diferentes características no desenvolvimento de um algoritmo de escalonamento de tempo real, dependendo o objetivo de cada projeto essas características podem estar presentes ou não. A Figura 5 mostra a hierarquia dessas características.

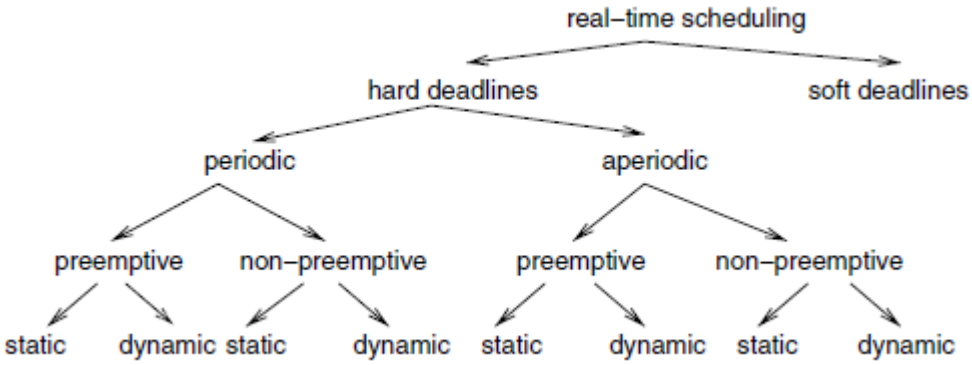

Figura 5 - Classes de algoritmos de escalonamento de tempo real.
Fonte: Marwedel (2006)

Como pode ser visto na Figura 5, Peter Marwedel (2006) define que um escalonador de tempo real pode assumir as seguintes características:

- *Hard deadlines*: existem muitas definições sobre as consideradas *hard deadline*, mas basicamente, entende-se como todas as tarefas que possuem um tempo rígido de execução. Considerando que se a tarefa não for comprido dentro do prazo é caracterizada uma falha de sistema.
- *Soft deadlines*: neste caso as tarefas possuem um certa flexibilidade nos seus tempos, sendo que é possível existir uma tolerância neste tempo. Quando ocorre uma falha de tempo neste caso, geralmente não é definido como uma falha e sim erro de qualidade de serviço. (Andrade e Oliveira, 2006)
- *Periodic*: ocorre sempre que as tarefas possuem um tempo pré-determinado de intervalo entre uma execução e outra.

- *Aperiodic*: as tarefas nesse caso não possuem um tempo pré-determinado para a execução, ou seja elas requerem o uso do processamento em momentos imprevisíveis.
- *Preempitive*: refere-se a quando uma tarefa pode ser interrompida antes de sua execução ter sido propriamente terminada, ou seja há uma interrupção na tarefa. Esse modelo talvez seja o mais comum entre os sistemas embarcados, visto que geralmente é necessário interromper as tarefas de prioridades menores, para liberar o processador para processar tarefas de maior importância (CARRO e WAGNER, 2003).
- *Non-Preemptive*: nesse tipo de escalonador, as tarefas nunca serão interrompidas durante sua execução.
- *Static*: essa característica significa que o escalonador não tomará decisões em tempo de execução. Ele apenas respeitará os tempos de partida das tarefas que foram definidos no *designer* da aplicação.
- *Dynamic*: nesse modelo, o escalonador pode tomar decisões sobre as tarefas em tempo de execução. Isso pode gerar problemas para o sistema, como *overhead* em tempo de execução.

1.3 Arduino

Uma revolução que vem acontecendo em sistemas embarcados no últimos anos é o surgimento do Arduino. Uma plataforma de prototipagem de sistemas embarcados utilizando uma placa única, sendo *open software* e *open hardware*. Os esquemas eletrônicos são liberados sob a licença CC-BY-SA (*Creative Commons Attribution Share-Alike)* e os códigos fontes, juntamente com suas bibliotecas, com licença LGPL (*Lesser General Public License*) (SITE Arduino). A Figura 6 mostra um modelo de placa Arduino.

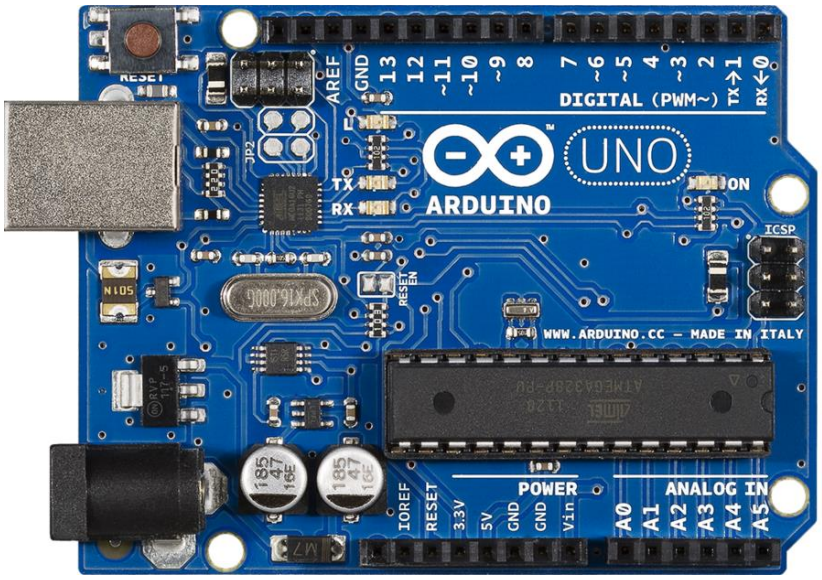

Figura 6 -Placa Arduino
Fonte: SITE Arduino

O projeto iniciou-se em 2005 na cidade de Ivrea na Itália, com o objetivo de integrar os projetos escolares, visando o menor custo na prototipagem. Desde então a plataforma já foi utilizada em milhares de projetos ao redor do mundo, desde projetos cotidianos e *hobista*, como em instrumentos científicos complexos (SITE Arduino).

Basicamente a plataforma Arduino pode ser dividida em duas partes, uma sendo o *hardware*, ou seja a placa utilizada, e a outra a IDE para a programação. O *hardware* Arduino é constituído basicamente por uma placa com um microcontrolador da família Atmel AVR, com os componentes básicos para o funcionamento, com cristais osciladores e barramentos seriais, juntamente com outros conectores que permitem a expansão dos projetos. Apesar do *hardware* ser baseado em Atmel, isso não limita a sua utilização. Isso devido a sua plataforma *open-source*, permitindo que placas 'compatíveis' sejam projetadas por terceiros, como é o caso da Funduino e Marminino, ambas versões brasileiras e compatíveis com a plataforma Arduino. O *software* utilizado na plataforma Arduino na maioria das vezes é codificada através da IDE da plataforma. A IDE utiliza uma pseudo linguagem baseada na linguagem tradicional C, como pode ser visto na Figura 7, encapsulando algumas configurações que a própria IDE se responsabiliza de realizar. A estrutura básica de *software* escrito na IDE Arduino pode ser visto na Figura 7.

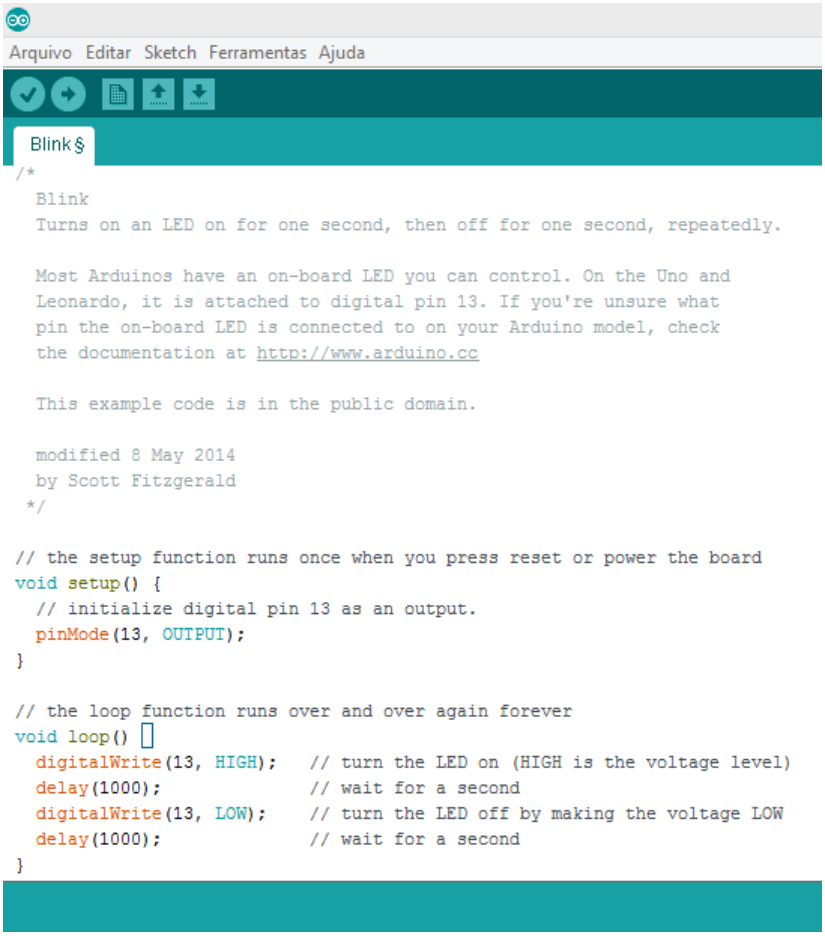

Figura 7 - Código Arduino
Fonte: AUTOR

2 AGENTES INTELIGENTES

Atualmente, no campo de pesquisa da Inteligência Artificial, não existe um consenso para a definição de um Agente Inteligente. Dentre as várias definições é possível destacar, que é um sistema computacional que está situado em um ambiente e é capaz de agir autonomamente sobre esse ambiente para que os objetivos de projeto sejam cumpridos (Wooldridge, 2002) e que, agente é um sistema computacional que está situado em algum ambiente e que é capaz de ações autônomas nesse ambiente em ordem de satisfazer seus objetivos (Russel e Norvig, 2003). Basicamente, é possível entender um agente inteligente como sendo uma entidade capaz de perceber seu ambiente e capaz de interagir de forma a realizar seus objetivos. Algumas outras características também são citadas por Wooldridge e Jennings (1995), como:

- Autonomia: agentes operarão sem a intervenção humana ou outra, eles tem controle sobre suas ações e seus estados internos.
- Habilidade social: agentes interagem com outros agentes e possivelmente humanos também.
- Reatividade: agentes percebem seu ambiente e respondem às alterações ocorridas.
- Pro-atividade: agentes não agem simplesmente por reação ao ambiente, eles também tem o comportamento de agir para um objetivo, tomando a iniciativa.

O ambiente em que o agente está situado também deve ser considerado, principalmente pois é sobre ele que suas percepções e ações são realizadas. Em Wooldridge (2002), os ambientes são classificados da seguinte forma:

- Acessíveis x Inacessíveis – um ambiente acessível é aquele onde o agente pode obter informações completas e precisas sobre o estado do ambiente. A maioria dos ambientes no mundo real não são acessíveis nesse sentido.
- Determinístico x Não-Determinístico – um ambiente determinístico é aquele em que cada ação tem um efeito único garantido – não há incerteza sobre o estado que irá resultar da realização de uma ação.
- Estático x Dinâmico – um ambiente estático é aquele que pode ser assumido como permanecendo inalterados, exceto pelo desempenho das ações do agente. Em

contraste, um ambiente dinâmico é aquele em que outros processos que operam nele, e que portanto, pode mudar de formas alheias a vontade do agente

- Discreto x Contínuo – um ambiente é discreto se houver um fixo, finito números de ações e percepções do mesmo.

2.1 Arquiteturas

A arquitetura, por vezes chamada tipo de raciocínio de um Agente, consiste no mecanismo lógico utilizado no seu desenvolvimento para que este possa interagir com o ambiente de forma inteligente. Em Wooldridge (2002) são definidos 4 tipos de arquiteturas, sendo elas, Dedutiva, Prática, Reativa e Híbrida. Algumas dessas arquiteturas, dependendo da literatura (Bellifemine, Caire e Greenwood, 2007), não possuem a mesma denominação, porém o mesmo conceito.

Dedutivo (ou simbólico): é considerada a abordagem 'clássica' de IA (Inteligência Artifical), onde um problema real é representado de forma simbólica pelo sistema, e baseada nessa representação o sistema realiza ações, utilizando-se das sequências lógicas em que foi codificado. O grande o problema nessa arquitetura é a dificuldade da representação simbólica do mundo real de forma precisa e descrição adequada, principalmente em ambientes mais dinâmicos.

Reativo: essa arquitetura é baseada no mapeamento direto entre um estímulo e a ação, não necessitando de uma representação simbólica complexa, como na Dedutiva. A teoria mais aceita nessa forma de arquitetura é a chamada subsunção de Brook (Wooldridge, 2002), que baseia-se em duas ideias principais, 'contextualização e personificação'. A inteligência 'real' está situada no mundo, não em teoremas; e 'Inteligência e emergência', onde o comportamento 'inteligente' é resultado da interação do agente com seu ambiente.

Híbrida (ou camadas): consiste em uma mescla das arquiteturas Reativa e Dedutiva, onde elas são organizadas em camadas, seja de forma horizontal ou vertical. Na horizontal, cada camada age como um agente, pois as entradas sensoriais estão diretamente ligadas com as saídas de ação. Já no conceito vertical, a entrada e saída são tratadas por mais de uma camada.

Prática: é o modelo de tomada de decisão onde o raciocínio é direcionado para ações, ou seja 'o processo de descobrir o que fazer' (Wooldridge, 2002), sendo o paradigma mais

aceito o BDI (*Belief-Desire-Intention*), pois é baseado no raciocínio prático proposto por Michael Bratman (1987). Na próxima sessão o tema é abordado mais profundamente.

2.1.1 Lógica BDI

A lógica BDI baseia-se na análise do comportamento definido por Bratman, (1987), onde é possível entender o comportamento humano em crenças, desejos e intensões. Para Bratman, a intenção e o desejo são as duas atitudes mentais ligadas com a ação, sendo a intenção mais ligado com o controle.

Bratman define que o comportamento ocorre devido a planos, sendo esses planos gerados por intenções do agente, onde um plano é um conjunto de ações e a intenção é um estado deliberativo em que o agente possui algum compromisso. A crença é entendida como o conhecimento prévio do agente, onde ele acredita em algo antes da experiência, já o desejo, define o objetivo, assim o agente é motivado a realizar os planos devido ao desejo.

Talvez a primeira arquitetura de agente a utilizar o paradigma BDI, também a mais durável até hoje, foi o chamado *Procedural Reasoning System* (PRS) ou sistema processual de raciocínio, originalmente desenvolvido em *Stanford Research Institute* por Michael Georgeff e Amy Lansky (Wooldridge, 2002). A Figura 8 ilustra o conceito de PRS.

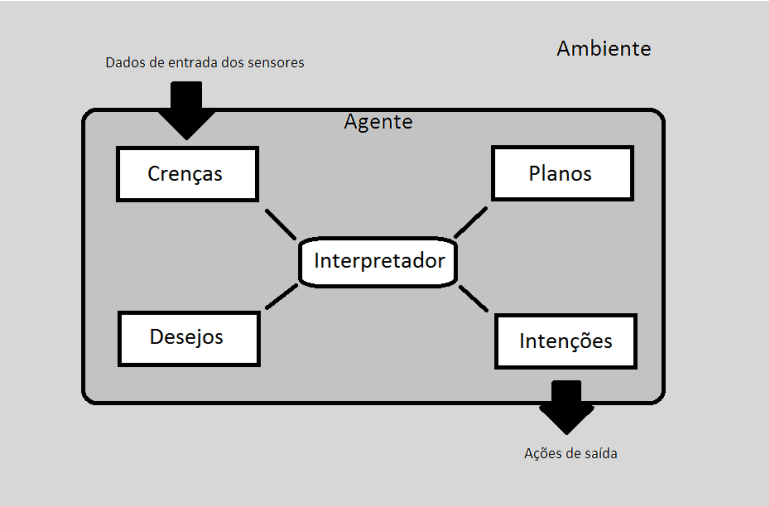

Figura 8: Procedural Reasoning System (PRS)
Fonte: Wooldridge (2002)

Na concepção de PRS, o agente é desenvolvido já com uma biblioteca de planos pré-compilados, assim o agente no seu estado inicial já possui uma lista de planos, sendo que

esses planos possuem três componentes, que são: uma pós-condição (objetivo do plano), uma pré-condição (contexto do plano) e um curso de ações a realizar (corpo do plano) (Wooldridge, 2002). Outro ponto que deve ser ressaltado nessa estrutura é o fato da existência do interpretador, que tem como função a atualização das crenças do agente, bem como realizar a deliberação dos planos. A deliberação, consiste na realidade, da escolha do melhor plano a ser seguido, sendo que em PRS, geralmente a escolha é feita pelo plano que o agente julga ter maior utilidade, caso o plano venha a falhar, o interpretador escolhe o próximo plano de sua lista e assim consecutivamente.

2.1.1.1 AgentSpeak (L)

Ao falarmos de Agentes DBI, é sobremaneira importante se descrever a AgentSpeak(L), uma das primeiras e talvez a mais conceituada das linguagens de programação baseada especificamente em programação de agentes BDI. A linguagem foi proposta por Anand S. Rao e apresentada a primeira vez em Rao(1996). AgentSpeak(L) foi projetada para a programação de agentes BDI na forma de sistemas reativos. Sistemas reativos são basicamente sistemas que estão em eterna execução, reagindo a eventos que ocorrem no ambiente com base em planos armazenados em sua biblioteca de plano. (HÜBNER, BORDINI e VIEIRA, 2004).

Um agente em AgentSpeak(L) consiste em um conjunto de crenças, esse conjunto é chamado de base de crenças. Cada crença por sua vez consiste em uma notação lógica usual. Em AgentSpeak(L) existem 2 tipos de objetivos, objetivos de realização e objetivos de testes. Os objetivos do agente expressam onde o agente pretende chegar. Na prática cada objetivo dispara uma sequência de subplanos para que esse objetivo seja alcançado. Um plano é formado por 3 elementos, um evento disparador, um contexto e um corpo. O evento disparador funciona como acionador de um plano. Assim quando o evento ocorre o plano se mostra apto a ser executado, contudo o contexto também deve ser verdadeiro para isso. O contexto consiste em expressões lógicas que devem ser verdadeiras para o plano. O contexto serve para evitar que o plano seja executado sem ter as condições necessárias para ter sucesso, isso faz por exemplo, que um plano que necessite de certas variáveis do ambiente, não seja executado até que essas variáveis se tornem promiscuas ao plano do agente. Já o corpo do plano consiste em ações que o agente vai realizar durante o plano, essas ações não são necessariamente, somente, mas podendo ser, ações no ambiente. Uma ação pode ser uma ação

interna, como um envio de mensagem, pode ser inclusive um subplano, que por sua pode executar mais uma série de ações.

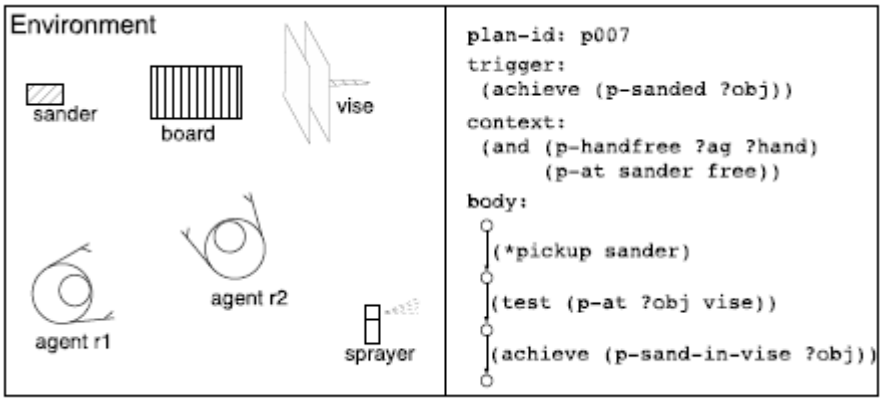

Figura 9 – Estrutura de um plano
Fonte: Guerra-Hernández e Ortiz-Hernández (2008)

A Figura 9 adaptada de Guerra-Hernández e Ortiz-Hernández (2008), ilustra um plano em AgentSpeak(L), a parte esquerda da Figura representa um ambiente fictício com 2 agentes, já a parte direita da Figura demostra um plano com a estrutura proposta pelo AgentSpeak(L), o plano é chamado de p007. Esse plano possui o evento ativador a percepção de um lixa, assim quando o agente perceber um lixa no ambiente, o plano p007 poderá ser acionado, para isso o contexto do plano também deve ser considerado verdadeiro. O plano p007, contém duas expressões em seu contexto, uma é a que sua mão deve estar livre `(and (p-handfree ?ag ?hand)` e que a lixa também esteja livre `(p-at sander free))`, caso as duas setenças sejam verdadeiras o corpo do plano poderá ser executado. O corpo do plano p007 possui 3 ações, a primeira é pegar a lixa `(*pickup sander)`, após isso ser realizado, o agente procura um torno `(test (p-at ?obj vise))` e por último coloca a lixa no torno `(achieve (p-sand-in-vise ?obj))`.

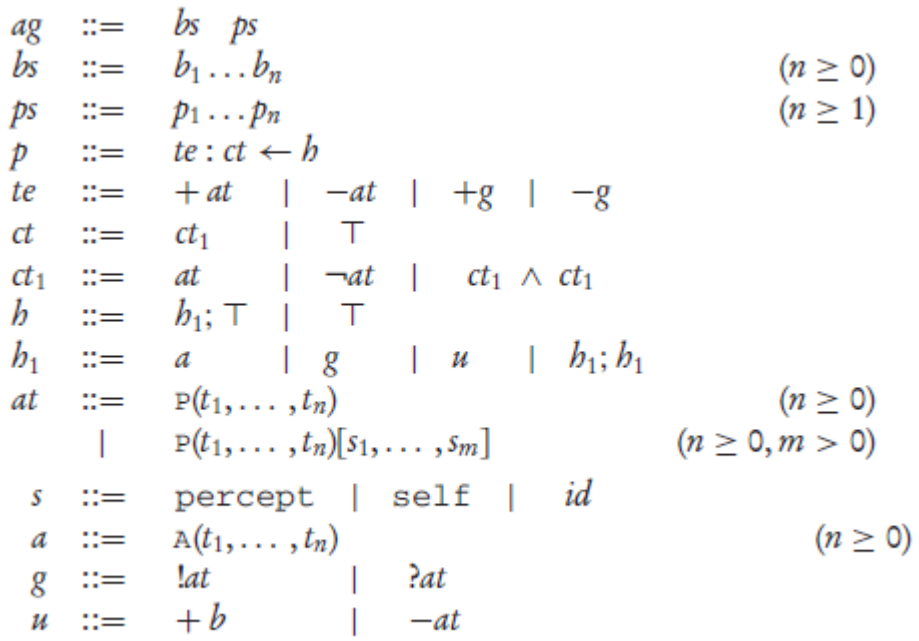

Figura 10 – Linguagem abstrata de AgentSpeak(L).
Fonte: Hübner, Bordini e Vieira (2004)

A linguagem abstrata de AgentSpeak(L) é mostrada na Figura 10, os elementos da linguagem possuem as seguintes representações:

- ag: é um agente especificado por um conjunto de crenças bs e um conjunto de planos ls.
- bs: é base de crenças do agente.
- ps: é biblioteca de planos do agente.
- at: são fórmulas atômicas da linguagem, que são predicados onde p é um símbolo predicativo.
- p: é um plano, formado por te, ct e h
- te: é o evento ativador, podendo ser at ou g.
- +at e –at: adição/remoção de crenças da base de crenças do agente .
- +g e -g : adição/remoção de objetivos.
- ct: é o contexto do plano.
- h: é uma sequência de ações, objetivos ou atualizações de crenças.

- u: corresponde a atualização da base de crenças.
- g: são os objetivos e podem ser de realização (!at) ou de teste (?at).

2.2 Comunicação

Outro tema muito importante dentro dos sistemas baseados em agentes é a comunicação entre os agentes, como afirma Karnouskos e Leitão(2012) ‘Agentes precisam de um meio de comunicação, a fim de ser capaz de cooperar’, isso se deve principalmente ao fato que, como citado anteriormente, uma característica fundamental do agente é a ‘habilidade social’.

Wooldridge (2002) cita a importância do estudo dos conceitos de comunicação na Ciência da Computação, e a sua grande diferença para o contexto do paradigma baseado em agente se comparado com outros paradigmas, como a orientado a objetos, por exemplo. O autor cita o exemplo:

> Suponha que temos um sistema Java contendo dois objetos, 01 e 02, e que 01 tem um método ‘ml’ com visibilidade pública. O objeto 02 pode se comunicar com 01 invocando o método ‘ml’. Em Java, isso significaria que 02 executa uma instrução que se parece algo como ‘01.ml(arg)’, onde ‘arg’ é o argumento de que 02 quer comunicar para 01. Mas considere: qual objeto toma a decisão sobre a execução de método ‘ml’? É objeto ou objeto 01 02? Neste cenário, o objeto 01 não tem controle sobre a execução de ‘ml’: a decisão sobre a eventual execução ‘ml’ encontra-se inteiramente com 02.
>
> Agora, considere um cenário semelhante, mas em um ambiente orientado a agentes. Temos dois agentes ‘i’ e ‘j’, onde ‘i’ tem a capacidade de executar a ação a que corresponde frouxamente a um método. Mas não existe um conceito no mundo orientado a agentes, de agente ‘j’ “chamar um método” de i. Isso ocorre porque ‘i’ é um agente autônomo: tem controle tanto sobre o seu estado e seu comportamento. Ele não pode ser dado como certo que o agente ‘i’ executará a ação só porque outro agente ‘j’ quer que ele o faça. A execução da ação pode ser que não seja um dos melhores interesses do agente ‘i’. O locus de controle com que diz respeito à decisão sobre se a executar uma ação é, portanto, muito diferente em sistemas de agentes e objetos. (Wooldridge, 2002, Traduzido pelo autor, p. 163,164)

Em (Karnouskos e Leitão, 2012) os autores dividem os tipos de comunicação entre agentes em dois tipos, sendo elas, a comunicação direta e a indireta. A comunicação direta é dita, quando existe uma troca direta de mensagens. Neste contexto, segundo os autores, as mensagens possuem um ‘envelope’ e um ‘conteúdo’, sendo que a mensagem em si é o conteúdo que está dentro do envelope. A comunicação indireta, é quando a comunicação não ocorre diretamente entre os agentes. Um exemplo de comunicação indireta, é o conceito de *blackboard.* Um agente posta a mensagem no quadro e todos os outros agentes podem acessa-las. Já em Huhns e Stephes (1993), é possível encontrar um estudo mais detalhado sobre as

características da comunicação entre agentes. Segundo os autores existem 3 aspectos no estudo formal da comunicação, que são a ‘sintaxe (como os símbolos de comunicação são estruturados), semântica (o que os símbolos denotam) e pragmática (como os símbolos são interpretados). Os autores ressaltam a importância do significado da mensagem, uma vez que o significado é a combinação de semântica e pragmática. Sendo que os agentes se comunicam a modo de compreender e serem compreendidos, torna-se assim fundamental o entendimento das dimensões que estão associadas ao significado. Huhns e Stephes (1993), descrevem essas dimensões.

- Descritivo X prescritiva: algumas mensagens servem para descrever fenômenos, enquanto outros prescrevem comportamentos.
- Significado pessoal X convencional: um agente pode ter o seu próprio significado de uma mensagem, sendo que esse significado pode ser diferente do significado convencional aceito pelos os outros agentes ao qual ele se comunica.
- Significado subjetiva X objetivo: semelhante ao significado convencional, em que o significado é determinado externo a um agente, uma mensagem, muitas vezes tem um efeito explícito sobre o meio ambiente, o que pode ser percebido objetivamente. O efeito pode ser diferente do que entendida internamente, ou seja, subjetivamente, pelo remetente ou destinatário da mensagem.
- Do orador X Do ouvinte X Perspectiva da sociedade: independente do significado convencional ou objetivo de uma mensagem, a mensagem pode ser expressa de acordo com o ponto de vista do orador ou do ouvinte ou dos outros membros da sociedade.
- Semântica X Pragmática: a pragmática da comunicação se refere a forma em que a comunicação será utilizada. Isto inclui considerações dos estados mentais dos comunicadores e do ambiente em que eles existem, essas considerações são externas a sintaxe e a semântica da comunicação.
- Contextualidade: mensagens não podem ser entendidas de forma isolada, mas devem ser interpretadas em termos dos estados mentais do agentes, o estado atual do ambiente, e da história do ambiente: como chegou a seu estado atual. As interpretações são diretamente afetadas pelas mensagens e ações anteriores do agente.
- Cobertura: linguagens menores são mais gerenciáveis, mas elas devem ser grandes o suficiente para que um agente possa transmitir o significados que pretende.

- Identidade: quando uma comunicação ocorre entre os agentes, o seu significado é dependente das identidades e papéis dos agentes envolvidos, e sobre a forma como os agentes envolvidos são especificados.
- Cardinalidade: uma mensagem enviada em particular para o agente é entendida de forma diferente do que a mesma transmissão realizada publicamente.

2.3 Sistemas Multi-Agentes

Existem várias definições para Sistemas Multi-Agentes (SMA), essa variação acontece devido a área em que o tema é abordado. Considerando SMA na área de arquitetura de *software*, proposta desse trabalho, é possível citar a definição feita por Shehory e Arnon (2014):

> 'Reconhecendo (SMA) como um estilo de arquitetura de *software*, SMA são sistemas que compreendem componentes chamados de agentes. Os agentes são normalmente concebidos para serem autônomos, onde autonomia refere-se a um componente não dependendo das propriedades ou dos estados de outros componentes para a sua funcionalidade.'

É possível entender assim um sistema multi-agentes como um sistema computacional formado por dois ou mais agentes inteligentes que interagem entre si e de forma independente, a fim de cumprir uma tarefa específica. Esse conjunto de agentes, de certa forma, gera uma sociedade, sendo a forma de organização dessa sociedade, uma das áreas mais estudadas em SMA. Shehory e Arnon (2014) citam as seguintes formas de organização em SMA: Hierárquica, Democrática (ou Plana), Subordinação, e Modular.

- Organização Hierárquica: neste tipo de organização, o controle é feito por agentes de níveis superiores, sendo eles responsáveis por a maioria das decisões ou não todas, assim é possível entender esse tipo de organização como centralizadora.
- Organização Democrática: nessa organização não existe o controle de um agente sobre o outro, sendo que o raciocínio do agente é formado com relação aos dos outros agentes, que faz a principal vantagem dessa organização ser o raciocínio dinâmico gerado entre os membros.
- Organização de Subordinação: essa organização ocorre quando um agente é formado por outros agentes. A forma de organização pode parecer com a Hierárquica, porém neste tipo, o agente componente é totalmente submisso ao contêiner que forma o outro agente. Essa organização também é similar ao modelo de Orientação a Objeto (OO),

porém a grande diferença é que na OO os objetos são ativados por chamadas definidas e já com Agentes, existe um alto nível de comunicação entre os membros.

- Organização Modular: neste tipo de organização, o sistema é divido em vários módulos, esses módulos podem ser gerados devido a divisões físicas ou quando é necessária a comunicação de vários agentes ou vários serviços.

3 FRAMEWORKS PARA SISTEMAS EMBARCADOS

Este capítulo realizaria uma introdução aos *frameworks* já existentes que seguem a mesma proposta do trabalho. Como durante a pesquisa não se encontrou exemplos de *frameworks* para sistemas embarcados utilizando a modelagem de agentes, se abordará, neste capítulo o desenvolvimento de sistemas convencionais baseados em agente. Como o objetivo principal do trabalho é o desenvolvimento do *software* dos sistemas embarcados e como o *hardware* é controlado pelo mesmo, não se adentrará no desenvolvimento convencional de sistemas embarcados, uma vez que todas as metodologias envolvem técnicas de especificações de *hardware*, não sendo esse o objetivo do trabalho.

3.1 Desenvolvimento de sistemas baseados em agentes

Segundo (Karnouskos e Leitão (2012)) existem, pelo menos 3 formas, de suporte para o desenvolvimento de sistemas baseados em agentes, sendo elas: metodologia de desenvolvimento, linguagens de programação baseada em agente, e *frameworks* e *toolkits* de desenvolvimento.

- Metodologia de desenvolvimento: uma metodologia consiste em um conjunto de métodos, modelos e técnicas a fim de facilitar o processo de desenvolvimento de um *software* em seu ciclo de vida completo. No contexto de sistemas baseados em agentes, se pode citar a participação da FIPA (*Foundation for Intelligent Physical Agents*). A FIPA é uma organização de padrões IEEE *Computer Society* que promove a tecnologia baseada em agentes e a interoperabilidade das suas normas com outras tecnologias (SITE FIPA). A FIPA tem como objetivo a especificação de protocolos para interação e plataforma externas do agentes, uma vez que os algoritmos de controle internos do agentes são difíceis de padronizar. Dentro das iniciativas da FIPA é possível a citar a AUML (A*gent Unified Modeling Language*) uma extensão baseada em agentes da conhecida UML, visando uma modelagem em larga escala para sistemas baseados em agentes. (Karnouskos e Leitão, 2012).
- Linguagens de programação baseada em agentes: geralmente chamada de AOP (*agent-oriented programming*). Como exemplo desse tipo de linguagem temos a AgentSpeak(L), visto no capítulo 2.1.1. Esse tipo de linguagem geralmente é formada por blocos básicos para projetar e implementar intencionais (Karnouskos e Leitão,

2012). Yoav Shoham (1993) descreve 3 principais componentes necessários em uma AOP, sendo elas:

- Uma linguagem formal restrita com a sintaxe e semântica clara para descrever um estado mental; o estado mental será definido exclusivamente por diversas modalidades, como crença e compromisso.
- Uma linguagem de programação interpretada para definir e programar agentes, com comandos primitivos, como 'REQUEST' e 'INFORM'; a semântica da linguagem de programação deve ser obrigatoriamente fiel com a semântica do estado mental.
- Um método para conversão de dispositivos neutros em agentes programáveis.

Yoav Shoham(1993) faz uma comparação das linguagens AOP com as OOP (object-*oriented programming*), como pode ser visto na Quadro 2.

Quadro 2 - OOP vs AOP
Fonte: Shoham(1993)

	OOP	AOP
Parâmetros de estado de definição da unidade básica.	Objeto sem restrições	Agente, crenças, compromissos, capacidades, escolhas ...
Processo de cálculo	Métodos de transmissão de mensagens e de resposta	Métodos de transmissão de mensagens e de resposta
Tipos de mensagens	Sem restrição	Informar, pedido, oferta, promessa, diminuir ...
Restrições sobre métodos	Nenhum	Honestidade, coerência

- *Frameworks* e *toolkits* de desenvolvimento: Karnouskose e Leitão (2012) realizam uma comparação de MAS com a arquitetura OSI de redes, uma vez que, os sistema baseados em agentes ou multi-agentes devem possuir camadas organizadas hierarquicamente, sendo as camadas mais baixas as responsáveis por tarefas mais básicas, sem necessidade de gerar inteligência para o agente. O *framework* mais popular atualmente para sistemas baseados em agentes é o JADE (*Java Agent*

DEvelopment framework), sendo uma das poucas ferramentas em conformidade com a FIPA. JADE é uma plataforma de *software* que fornece funcionalidades *middleware* da camada de base que são independentes da aplicação específica e que simplificam a realização de aplicações distribuídas que exploram a abstração do agente em *software* (Bellifemine, Caire E Greenwood, 2007). É possível citar 3 características básicas de *designer* de um agente utilizando o *framework* JADE (Bellifemine, Caire E Greenwood, 2007):

- Um agente é autônomo e pró-ativo: um agente não pode fornecer *call-backs*, nem seu objeto pode ser passado por referência a outros agentes. Somente o próprio agente pode controlar seu ciclo de vida e decidir quando realizar qualquer ação.
- Os agentes podem dizer 'não', e eles são fracamente acoplados: a comunicação é baseada em mensagens assíncronas. Quando um agente quer fazer uma comunicação é necessário que ele envie uma mensagem para um destinatário identificado, sendo que não existe nenhuma dependência entre o emissor e receptor da mensagem. O receptor por sua vez, processa as mensagens recebidas conforme a prioridade que ele mesmo estabelece, sendo possível, inclusive, que ele descarte a mensagem sem processa-la.
- O sistema é *Peer-to-Peer*: como definido pela FIPA, cada agente possui um nome exclusivo, ou AID (*AgenteIDentifier*). Um agente pode realizar a comunicação com qualquer outro agente, sendo que é possível que ele realize uma consulta em seu catálogo, e verifica qual é o identificar do agente destino, igualmente outros agentes podem começar uma troca de mensagens com ele da mesma forma.

3.2 Proposta de Framework

Após analisados todos os aspectos vistos neste trabalho, é possível propor um modelo de *framework* de desenvolvimento de agentes inteligentes em um sistema embarcado. Essa implementação traz uma série de vantagens, como: padronização de um modelo de desenvolvimento, trazendo mais agilidade ao processo; encapsulamento das funcionalidades mais básicas diretamente no *framework*, evitando assim retrabalho em outros projetos; abstração do funcionamento básico de *hardware*, assim não havendo necessidade do

desenvolvedor conhecer o código de baixo nível, como proposto por Karnouskose e Leitão (2012).

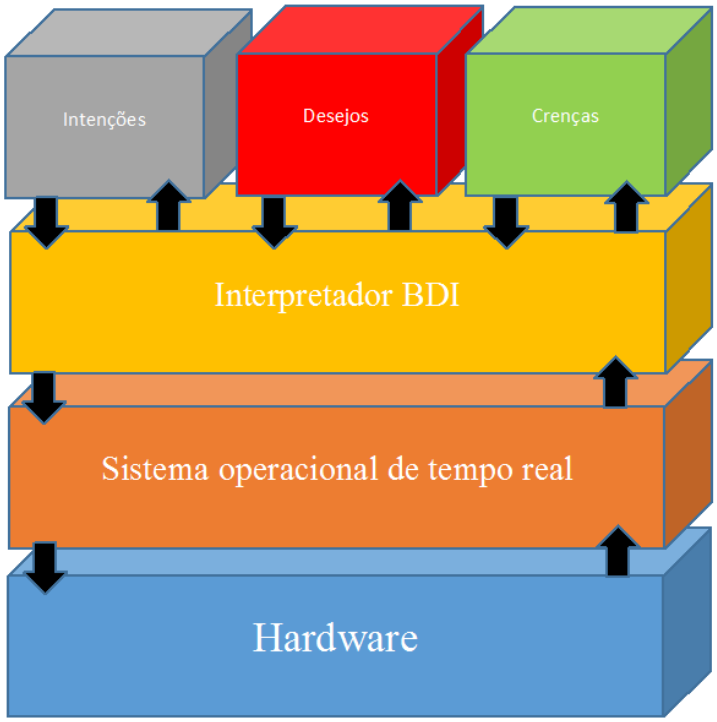

Figura 11: Organização funcional do *framework* proposto
Fonte: AUTOR

Como pode ser visto na Figura 11, o sistema proposto está dividido em 4 camadas, como sugerem Karnouskose e Leitão (2012), gerando uma arquitetura parecida com a defendida por Tammy Noergaard (2012), visando o maior nível de abstração entre as camadas como proposto por Nebel e Schumacher (1996). As camadas são: a de *hardware*, sistema operacional de tempo real, interpretador BDI e a camada de dados.

A camada de *hardware* é constituída da placa base do sistema, juntamente com os sensores e atuadores do agente, sendo ela a interface do sistema com o ambiente real, bem como, responsável pela transmissão e recepção de dados entre outros agentes.

A segunda camada é onde está situado o sistema de tempo real. Essa camada será responsável por suportar a terceira camada e interligar ela com a primeira camada, assim abstraindo totalmente o *hardware* da aplicação. A escolha de um sistema operacional de tempo real para a camada 2, se deve ao fato de que essa camada é responsável por monitorar o *hardware* e gerar a noção de ambiente para as camadas superiores. Para isso a criação de

tarefas é fundamental, pois qualquer falha ou demora no monitoramento externo pode gerar decisões erradas do agente e assim afetar seu desempenho no ambiente. Outro fato que foi levado em consideração para a escolha de um sistema de tempo real é a sua facilidade de modularização, pois além do *hardware* ele deve executar com precisão o interpretador BDI que está situado na camada superior.

A terceira camada é a camada do interpretador BDI. Essa camada é basicamente a camada da aplicação e tem como função interpretar os estados do agentes, que estão situados na quarta camada. Ela deve operar como proposto no sistema PRS, sendo responsável pela manutenção das crenças do agente e pela deliberação dos planos para a conclusão das intenções do agente, afim de conseguir concluir o objetivo esperado, bem como ser capaz de formar uma crença sobre o estado do objetivo em questão.

A quarta camada, pode ser considerada a camada de dados, pois é nessa camada onde ficam os dados relativos ao comportamento do agente, como suas crenças. Essa camada deve ser codificada de forma que o interpretador possa acessar seus dados e transmiti-los para as camadas inferiores, assim gerando as ações e deliberações do agente.

3.3 Funcionamento

A proposta tem o objetivo de tentar abstrair o quanto possível as camadas mais inferiores da aplicação, sendo que, caso o desenvolvedor queira, ele possa desenvolver um agente sem que seja necessário codificações de linguagens de baixo nível. Para isso é necessário que esses estados sejam codificados utilizando uma linguagem que o interpretador entenda. Como visto no Quadro 1 de Noergaard (2012), os sistemas embarcados não suportam a quinta geração de linguagens de programação, como é o caso das linguagens orientadas a agente, neste contexto apresenta-se aqui o primeiro desafio na construção da proposta.

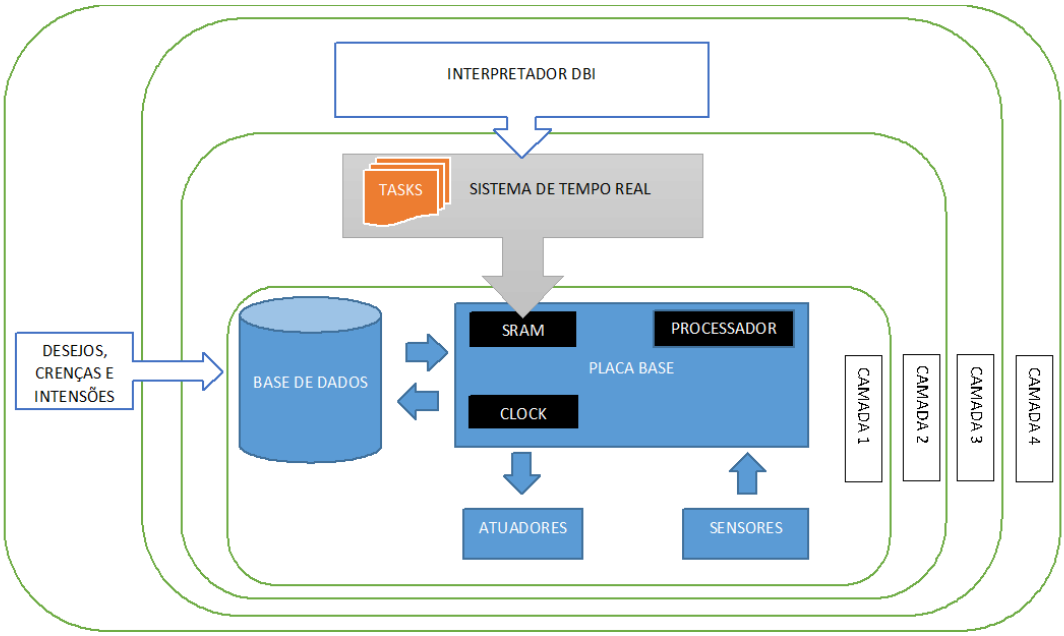

Figura 12: Funcionamento do *framework* proposto
Fonte: AUTOR

A Figura 12 demonstra o funcionamento teórico do *framework*, onde pode ser visto que na camada 1 estão todos os componentes de hardwares do sistema, como os sensores, atuadores, dispositivos de armazenamento de dados e a placa base. Nessa placa base devem existir pelo menos 3 componentes básicos, sendo eles o *clock*, a memória não volátil e o processador. O *clock* é necessário para o sistema de tempo real da camada superior, a memória não volátil é necessária pois é nela que o sistema de tempo real ficará gravado e o processador para que o sistema possa ser executado. Portanto é possível entender a camada de *hardware* como sendo um microcontrolador juntamente com outros periféricos. A camada 2 é formada apenas pelo sistema de tempo real, esse sistema deve possuir no mínimo 4 tarefas em execução. A camada 3 consiste apenas no interpretador BDI, esse interpretador é formado por uma tarefa principal em execução. Porém como dito, existem mais 4 tarefas em execução. Essas tarefas serão explicadas ao decorrer do trabalho. O funcionamento do interpretador consiste em consultar a base de dados, deliberar sobre os estados e atualizar a situação dos estados quando necessário. Já a camada 4 consiste praticamente em uma base de dados, onde estão todos os dados variáveis do sistema, como por exemplo sua base de crenças.

4 IMPLEMENTAÇÃO DO FRAMEWORK

4.1 Sistema operacional de tempo real FreeRTOS

Para o desenvolvimento da camada do sistema operacional do *framework* proposto, foi escolhido o FreeRTOS. A escolha foi devido principalmente, a ser um sistema de código aberto e uso livre, também devido a ser um dos RTOS mais utilizados na atualidade. Por tanto, as sessões a seguir, tem como propósito realizar um breve resumo do funcionamento desse componente do *framework*.

O FreeRtos foi desenvolvido por Richard Bary e hoje é mantido pela empresa Real Time Engineers. Foi escrito na linguagem C, sendo de uso livre e de código aberto. Existem algumas variações do sistema, devido a licença de uso, um exemplo é OpenRTOS, que possui o mesmo código fonte, porém com suporte e certificações.

O FreeRTOS segue o conceito de *super-loop* para a criação das *threads*. Assim cada *threads* na verdade é constituída de um programa em C contendo um laço infinito, que será executado até que haja uma interrupção. Essa interrupção pode ser ocasionada pela própria *thread* ou então pelo escalonador do sistema, que liberará o processador para outra *thread* ser executada. Mais detalhes do funcionamento do sistema podem ser vistos nos próximos subcapítulos, sendo todos eles baseados na obra escrita por Bary(2009).

4.1.1 Configuração do sistema

Como o FreeRtos é um sistema flexível e por ser utilizado o mesmo *kernel* para diversas plataformas, as suas configurações de funcionamento deve ser feita a cada projeto. Isso é configurado no arquivo *Header* chamado FreeRtosConfig.h, que por sua vez, deve ser adicionado a cada projeto. Este capítulo explicará os principais parâmetros que devem ser configurados no sistema FreeRtos.

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

#include <p18cxxx.h>

#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK             0
#define configUSE_TICK_HOOK             0
#define configTICK_RATE_HZ              ( ( portTickType ) 1000 )
#define configCPU_CLOCK_HZ              ( ( unsigned long ) 20000000 )
#define configMAX_PRIORITIES            ( ( unsigned portBASE_TYPE ) 4 )
#define configMINIMAL_STACK_SIZE        ( 105 )
#define configTOTAL_HEAP_SIZE           ( ( size_t ) 1024 )
#define configMAX_TASK_NAME_LEN         ( 4 )
#define configUSE_TRACE_FACILITY        1
#define configUSE_16_BIT_TICKS          1
#define configIDLE_SHOULD_YIELD         1

#define configUSE_CO_ROUTINES       0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

#define INCLUDE_vTaskPrioritySet        0
#define INCLUDE_uxTaskPriorityGet       0
#define INCLUDE_vTaskDelete             1
#define INCLUDE_vTaskCleanUpResources   0
#define INCLUDE_vTaskSuspend            1
#define INCLUDE_vTaskDelayUntil         1
#define INCLUDE_vTaskDelay              1


#endif
```

Figura 13. Exemplo de configuração do FreeRTOS
Fonte: AUTOR

Na Figura 13 pode ser visto um exemplo de configuração do sistema, retirada de um dos exemplos disponibilizados junto do FreeRTOS. O primeiro parâmetro importante visto no exemplo acima é o "#include <p18cxxx.h>", esse parâmetro faz a referência à biblioteca da do microcontrolador, irrelevante neste momento. Os parâmetros seguintes são mais estritamente ligados com o funcionamento do sistema. Os parâmetros onde são atribuídos valores sem o uso de parênteses "( )", apenas deve ser passados valores 1 ou 0, sendo "sim" e "não" respectivamente, como é o caso do parâmetro "configUSE_PREEMPTION". Esse parâmetro define como o escalonador irá operar, caso receba 1, como o nome já diz, o escalonado irá operar de forma preemptiva. Caso receba 0 o escalonador do sistema irá trabalhar de forma colaborativa. Na forma preemptiva, como visto no capítulo 1.2.1, o sistema irá interromper as tarefas de forma periódica, para verificar se existe outra tarefa de igual ou superior prioridade pronta a ser executada. Na forma colaborativa não existe interrupções por parte do escalonador, e sim somente pela própria tarefa ou por outra tarefa.

Os próximos 2 parâmetros fazem referência ao *HOOK* ou gancho. Caso esteja atribuídos o valor 1 nesses parâmetros é possível fazer uma chamada "gancho". No caso do parâmetro "configUSE_TICK_HOOK", ele fará que o escalonador chame a tarefa gancho a cada troca de contexto ou tarefa. No caso do parâmetro "configUSE_IDLE_HOOK" o sistema irá chamar a tarefa durante a chamada tarefa "*IDLE*" (ou ociosa). Como o escalonador do

FreeRtos sempre deve ter uma tarefa sendo executada, o sistema cria uma tarefa falsa chamada de "*IDLE*" para manter o escalonador ocupado.

Os parâmetros "configTICK_RATE_HZ" e "configCPU_CLOCK _HZ" são de velocidade, o primeiro é usado quando o sistema opera de forma preemptiva, e define a velocidade de cada interrupção, o segundo é usado para a definição da velocidade que o processador vai operar. O "configMAX_PRIORITIES" defini até onde vai a escala de prioridade sendo o 0 o nível mais baixo.

4.1.2 Task ou tarefas

No FreeRtos cada *thread* de execução é chamada de "*task*" ou tarefa em português. Uma *task* consiste em uma função escrita em C que deve seguir 2 regras básicas: deve conter uma laço infinito, geralmente um laço "for(;;)"; e que uma *task* mesmo sendo uma função, não deve retornar valor algum, ou seja nunca deve ser implementado a chamada "return". Por isso as *tasks* sempre utilizam a assinatura "void". Um protótipo de uma *task* pode ser visto na Figura 14.

```
ATaskFunction vazio (void * pvParameters)
{
        for (;;)
        {
        }
        vTaskDelete (NULL);
}
```

Figura 14. Protótipo de uma *task*.
Fonte: AUTOR

Mesmo que uma *task* esteja criada, não necessariamente ela vai estar em execução no escalonador, a *task* só entra em execução quando ocorrer o chamado da função "xTaskCreate()". Essa função é considerada a mais importante do sistema pois ela é a responsável por adicionar as tarefas ao escalonador. A função "xTaskCreate()" possui a seguinte assinatura "*xTaskCreate(pvTaskCode, pcName, usStackDepth, pvParameters, uxPriority, pxCreatedTask)*", sendo:

- pvTaskCode: o nome da *task* que será adicionada ao escalonador, ou seja o nome da função, no caso da Figura 14 por exemplo, seria "A*Task*Funcion".
- pcName: um apelido para a tarefa, apenas serve para auxiliar algum tipo de debug. O tamanho do nome não é livre, e sim deve ser configurado no parâmetro "configMAX_TASK_NAME_LEN" presente no arquivo de configuração do FreeRTOS.
- usStackDepth: esse parâmetro serve para configurar quanto de memória RAM a *task* poderá utilizar do sistema. O parâmetro "configMINIMAL_STACK_SIZE", presente no arquivo de configuração do sistema, define qual o menor valor a ser atribuído a uma tarefa qualquer.
- pvParameters: qualquer *task* pode receber um ponteiro de uma variável, sendo esse valor atribuído ao parâmetro "pvParameters".
- uxPriority: esse parâmetro define qual é o nível de prioridade da *task*, esse valor deve estar entre 0 e o parâmetro "configMAX_PRIORITIES".
- pxCreatedTask: esse parâmetro pode ser utilizado para adicionar uma referência a tarefa, ou seja quando é necessário fazer alguma operação na tarefa, como por exemplo alterar sua prioridade, se faz necessário utilizar esse parâmetro. Caso não seja necessário fazer nenhuma chamada a essa tarefa se pode passar o valor NULL a esse parâmetro.

Uma vez feita a criação da *task*, ou seja a chamada da função "xTaskCreate()", é possível o retorno de 2 valores diferentes, informando se a tarefa foi criada com sucesso ou não. Os valores podem ser:

- pdTRUE: *Task* criada com sucesso.
- errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY: Não foi possível criar a *task*, pois houve falta de memória Heap.

4.1.3 Estados de uma *task*

Na Figura 14 é possível ver os possíveis estados de um *task*, sendo eles Suspended(suspenso), Ready(pronto), Blocked(bloqueada) e Running(execução).

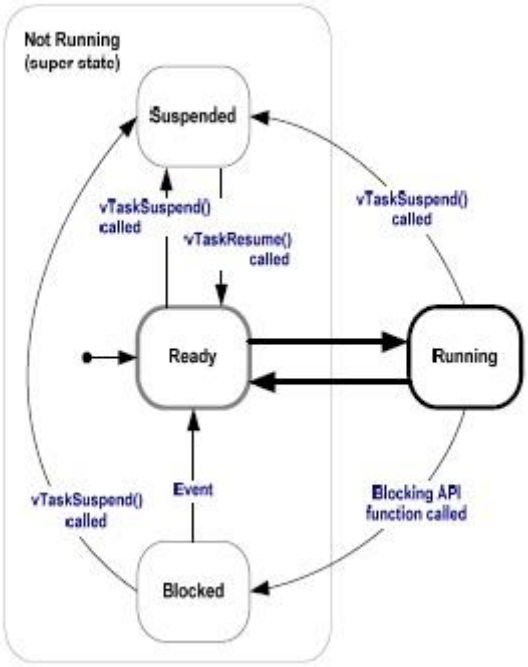

Figura 15. Diagrama de troca de estados de um *task*.
Fonte: Bary (2009)

- Blocked (bloqueada): uma *task* está nesse estado quando necessita de algum evento para poder ser executada, esses eventos podem ser de dois tipos básicos, eventos temporais, que são tempos de "espera" que a tarefa deve aguardar e o outro tipo de evento, são os chamados eventos de sincronização, esses eventos são ocasionados por outra tarefa ou uma função de interrupção, os eventos de interrupção podem ser de diversas formas, sendo os mais conhecidos as filas e os semáforos binários.
- Suspended (suspenso): o estado suspenso faz com que a *task* esteja totalmente indisponível, a única forma de uma *task* entrar estado é a chamada da função o "vTaskSuspend()", e a única forma de sair é a chamada da função "vTaskResume()".
- Ready (pronto): a *task* estará nesse estado sempre em que a tarefa esteja pronta para ser executada, ou seja não dependendo de nenhum evento e sim apenas que seja escolhida pelo escalonador.
- Running(execução): A *task* está em execução no processador.

4.1.4 DuinOS

DuinOS é uma versão do FreeRTOS desenvolvida especificamente para ser compatível com as bibliotecas Arduino, bem como a IDE Arduino. Criado em 2009 por Julián U. da Silva Gillig, o DuinOS consiste no núcleo do FreeRTOS e as bibliotecas do *kernel* do Arduino embutidos em um único projeto. O projeto pode ser utilizado diretamente na IDE Arduino, como em outras IDEs, como será visto mais adiante neste trabalho. A utilização do DuinOS traz uma série de vantagem para desenvolvedores que querem trabalhar com as placas Arduino ou com a família de microcontroladores AVR. O que dificulta a popularização do sistema é aparente a falta de interesse pelo sua utilização, talvez isso se deva pela falta de documentação existente no projeto, contanto basicamente como o material disponível no GitHub do projeto (Projeto DuinOS).

4.2 Interpretador BDI

Antes de iniciar-se a implementação do interpretador do *framework* proposto, se faz necessário realizar uma revisão bibliográfica sobre o assunto, bem como analisarmos modelos já existente e conceituados desse tipo de interpretador. Por tanto, os capítulos a seguir tem por objetivo realizar uma revisão sobre os interpretadores BDI.

Na literatura atual é possível encontrar algumas propostas de interpretadores utilizando o conceito BDI, visto no capítulo 2. Shohan (1993) define um interpretador básico com 2 passos essenciais, sendo eles:

> (1) Ler as mensagens atuais e atualizar o seu estado mental, incluindo suas crenças e compromissos do programa do agente é crucial para esta atualização;
>
> (2) Executar os compromissos de tempo atual, possivelmente, resultando na mudança de crenças (esta fase é independente do programa do agente). (Shohan, 1993).

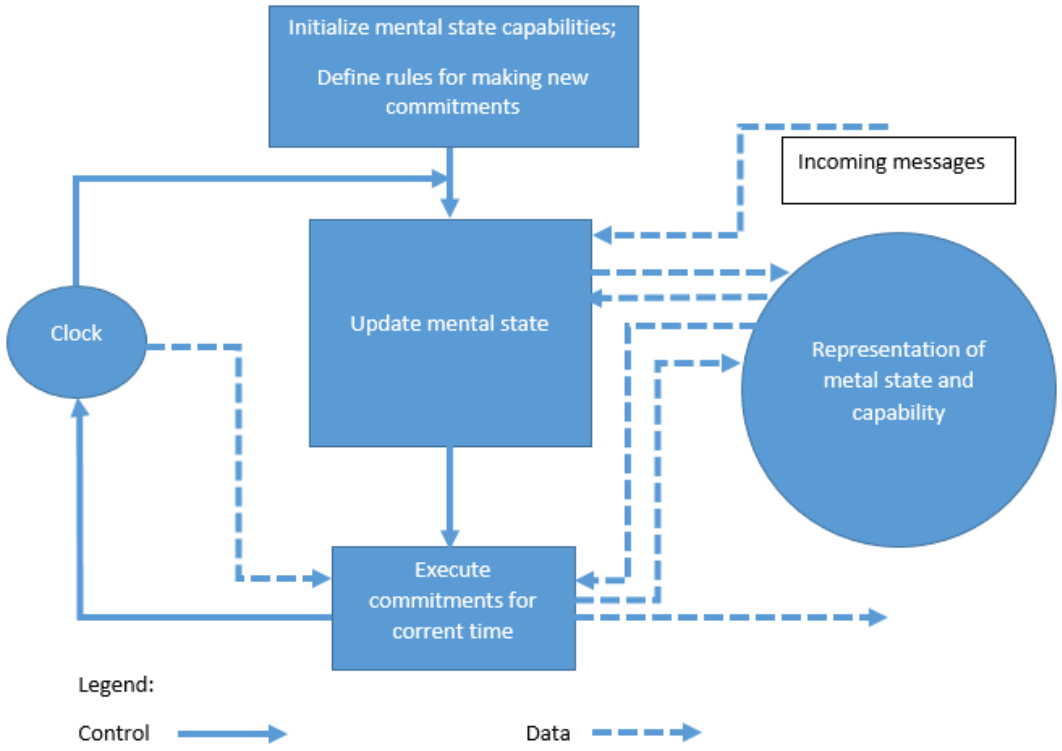

Figura 16 – Interpretador básico
Fonte: Shohan (1993)

A Figura 16 demonstra a estrutura do interpretador básico. Shohan (1993) ainda define algumas premissas na especificação do interpretador, sendo elas:

- Envio de mensagem: é subentendido que a plataforma é capaz de realizar o envio e o recebimento de mensgens entre agentes, sendo que a linguagem de programação do agente deve definir a estrutura da mensagem e o interpretador fica responsável apenas de realizar o envio da mensagem no momento correto.
- Relógio: a utilização de um relógio é essencial na contrução do interpretador, isso se deve ao fato que deve existir ciclos de repetição no interpretaodor, e para que isso ocorrá é necessária a utilização de um relógio ou algo que de um noção de intervalos que possam disparar as funções que estão dentro do *loop* central.

O modelo de interpretador apresentado em Shohan (1992) é bem simplório, podendo ser considerado como um modelo não prático, uma vez que não consegue atender exigências de tempo real colocadas sobre o sistema (GEORGEFF, RAO e SINGH ,1999).

Outra proposta, já bem mais detelhada, é apresentada em Georgeff, Rao e Singh (1999). Os autores descrevem um interpretador básico, porém utilizando os conceitos

adjacentes dos agentes DBI, interpretador que por sua vez, possui a essência do PRS, apresentado no capítulo 2. A Figura 17 apresenta o pseudo código básico desse interpretador. Não entra-se em detalhes sobre o funcionamento do código proposto pelos autores, no entanto suas principais funções são explicadas nos parágrafos a seguir.

```
BDI-interpreter
initialize-state();
do
    options := option-generator(event-queue,B,G,I);
    selected-options := deliberate(options,B,G,I);
    update-intentions(selected-options,I);
    execute(I);
    get-new-external-events();
    drop-successful-attitudes(B,G,I);
    drop-impossible-attitudes(B,G,I);
until quit.
```

Figura 17 – Interpretador BDI
Fonte: Georgeff, Rao e Singh (1999)

No funcionamento proposto pelos autores, as entradas externas ocorrem através de eventos, que são armazenados numa fila, formando uma fila de eventos, sendo todos esses eventos considerados atômicos. As saídas do sistema são determinadas como ações e também são consideradas atômicas, podendo ou não reconhecer os eventos correnspondentes ao sucesso da ação. Assim com base no estado atual e os eventos armazenados na fila, o sistema seleciona e executa opções, que correspodem a sub-rotinas.

Segundo a proposta dos autores, todos os eventos que ocorrem externamente ou internamento ao agente, são colocados em uma fila. A cada ciclo do interpretador a lista de eventos existente é submetida a uma função chamada '*option-generator*' ou gerador de opções. Essa função tem como propósito verificar a biblioteca de planos do agente, quais planos tem como base o evento como disparador. Quando um evento condiz com um plano existente, a função deve verificar se as pré-condições do plano são atendidas, caso sejam o plano é colocado na lista de planos que será retornada pela função. A Figura 18 demonstra um pseudo código proposto pelos autores para a função de geração de opções.

```
option-generator(trigger-events)
options := {};
for trigger-event ∈ trigger-events do
    for plan ∈ plan-library do
        if matches(invocation(plan),trigger-event) then
            if provable(precondition(plan),B) then
                options := options U {plan};
return(options).
```

Figura 18 - Gerador de opções
Fonte: Georgeff, Rao e Singh (1999)

Outro ponto importante no modelo proposto pelos autores é a chamada da função *'deliberate'* ou deliberar. Uma vez que a função '*option-gerator*' pode resultar numa lista de planos e tendo como premissa que o agente só pode implementar uma plano por vez, se faz necessário que ocorra uma deliberação sobre qual plano irá ser realizado. Para isso essa função escolhe algum dos planos com base em informação de meta nível, ou seja, com base numa crença de prioridade do agente, caso não seja possível definir um plano prioritário, a função simplesmente escolhe aleatóriamente um plano. A Figura 19 representa o pseudo código proposto pelos autores para uma função de deliberação:

```
deliberate(options)
if length(options) ≤ 1 then return(options);
else metalevel-options := option-generator(b-add(option-set(options)));
    selected-options := deliberate(metalevel-options);
    if null(selected-options) then
            return(random-choice(options));
slse return(selected-options).
```

Figura 19 – Função de deliberação
Fonte: Georgeff, Rao e Singh (1999)

Uma vez escolhido o plano, o restante que o interpretador deve fazer é executar o plano e atualizar seus estados mentais ocasionados pela execução do plano.

Uma implementação prática da proposta feita pelos autores Georgeff, Rao e Singh (1999), vista até agora, é o interpretador Jason, que utiliza a linguagem AgentSpeak(L), visto no capítulo 2.2.1.1. Não entra-se em detalhes sobre a ferramenta Jason, porém ela pode ser descrita como um interpretador para uma extensão de AgentSpeak incluindo comunicação entre agentes baseada na teoria de atos de fala (BORDINI, HÜBNER e VIEIRA, 2004).

Sendo assim, o estudo sobre o Jason é direcionado apenas ao seu interpretador, pois esse é o objetivo neste momento.

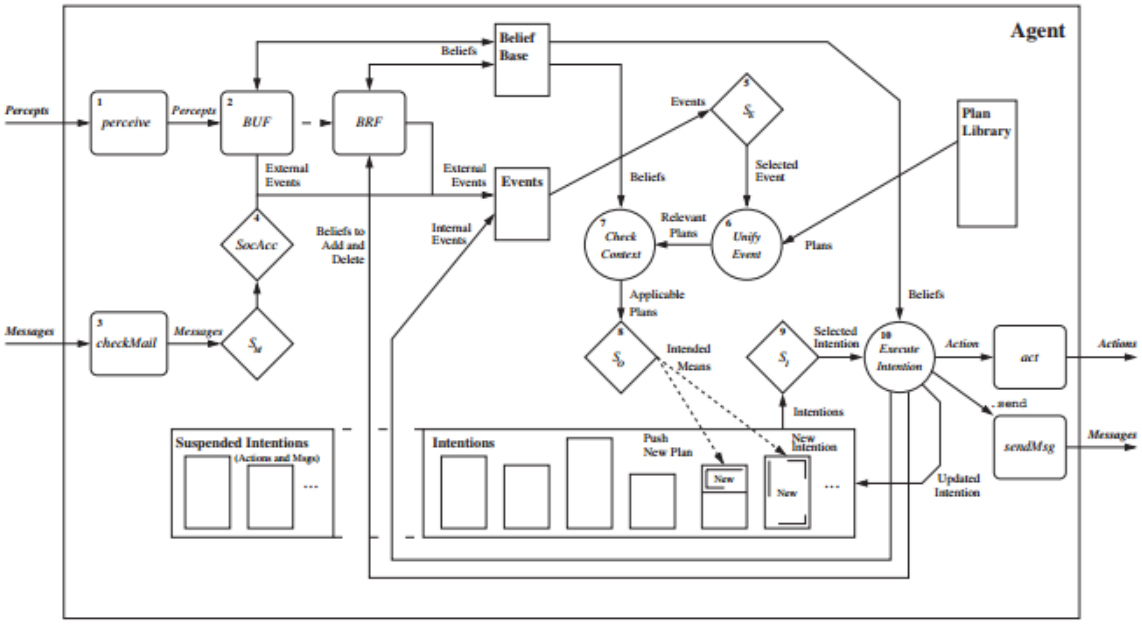

Figura 20 – Interpretador Jason.
Fonte: Bordini, Hübner e Wooldridge (2007)

A Figura 20 representa o funcionamento do interpretador Jason, apresentado por Bordini, Hübner e Wooldridge (2007). Na Figura é possível ver diferentes formas geométricas, cada forma geométrica é representada com as seguintes características (BORDINI, HÜBNER e WOOLDRIDGE, 2007):

- Retângulos: representam os principais componentes da arquitetura que determinam o estado do agente, como a base de crenças, o conjunto de eventos, a biblioteca da planos e o conjunto de intenções.
- Caixas arredondadas: representam funções que podem ser personalizadas no interpretador.
- Losangos: igualmente as caixas arredondas podem ser personalizadas, porém são diferenciadas, pois são funções de seleção, que recebem um lista e decidem a utilização de um deles.
- Círculos: são funções estáticas do interpretador que não podem ser personalizadas pelo programador.

Bordini, Hübner e Wooldridge (2007) subdividem o ciclo de raciocínio do interpretador Jason em 10 sub etapas, que podem serem vistas na Figura 20, com o índice correspondente. São elas:

1- Verificação do ambiente: como visto no capítulo 2, uma das premissas do conceito de agente inteligente, é possibilidade do agente poder interagir com o ambiente, tanto de forma receptiva como de forma atuante. Assim, o primeiro estágio do interpretador é realizar a verificação do ambiente e interpretar as mudanças relevantes ao agente.

2- Atualização das crenças sobre o ambiente: uma vez sentida uma mudança em alguma variável do ambiente no passo 1, o passo 2 é responsável por atualizar a crença do agente sobre aquela variável do ambiente.

3- Verificação de mensagens: outra premissa também discutida no capítulo 2, é a importância do agente conseguir se comunicar com outros agentes. Para isso acontecer, o passo 3 é responsável por verificar se existe alguma mensagem a ser recebida ou até enviado pelo agente.

4- Seleção de mensagens: esta etapa tem o objetivo de avaliar se as mensagens recebidas podem ou não ser aceitas pelo agente. Essa etapa é chamada de processo de aceitação social, sendo que deve ser personalizada para cada agente.

5- Seleção de um evento: uma vez alcançado essa etapa o agente deve possuir um ou vários eventos em sua fila de eventos, eventos ocasionados pela etapa 3 e 5, porém o interpretador só executa um evento por ciclo, por isso a etapa 5 tem o objetivo de analisar e selecionar apenas um evento para ser processado. Essa escolha é realizada com base nas prioridades e meta dados existentes na personalização dessa função.

6- Seleção de planos para o evento: ao passo que o interpretador já selecionou o evento escolhido, agora se faz necessário realizar a busca por todos os planos existentes que podem ser disparados com o acontecimento do evento.

7- Verificação das pré-condições dos plano: conforme visto no parágrafo anterior, Georgeff, Rao e Singh (1999) defendem que todo plano possui, além de um evento de disparo, também pré-condições que devem ser atendidas para plano ser aceito como válido. Isso acontece neste passo 7 do interpretador do Jason, ela é responsável por avaliar quais planos podem ser executado com base nas suas pré-condições.

8- Selecionar o plano mais relevante: esta etapa também é uma função de seleção, uma vez que a etapa anterior retorna todos os planos considerados como validos para o processamento do evento ocorrido. Esta etapa deve retornar o plano mais relevante entre os planos escolhidos. Esta seleção ocorre com base em meta informações personalizadas para cada agente.

9- Seleção da intenção: cada plano é formado por 3 componentes básicos, como visto no capítulo 2.2.1.1, um evento disparador, pré-condições e um conjunto de intenções. O conjunto de intenção pode ser definido com sub-planos a serem executados. O interpretador, uma vez que o plano foi selecionado, só pode executar uma intenção por vez, sendo assim o passo 9 tem como objetivo escolher o ação a ser executada.

10- Execução do intenção: a última etapa do ciclo de raciocínio do interpretador é a execução da intenção selecionada na etapa anterior.

Uma vez analisadas alguns desses interpretadores é possível realizar a proposição de uma solução. Primeiramente é valido lembrar, que um *software* embarcado possui métricas de eficiências diferentes de um *software* convencional, como mencionado no capítulo 1.É importante lembrar também que o código é literalmente embarcado no *hardware*, como visto no capítulo 1.2. Assim a solução deve possuir o mínimo de código possível e gerar o mínimo possível de variáveis, uma vez que essas variáveis ocupam a memória SRAM do microcontrolador, que por sua vez é muito menor que a memória FLASH (memória onde fica os códigos estáticos). A Figura 21, mostra o fluxo de ciclo de raciocínio do interpretador proposto.

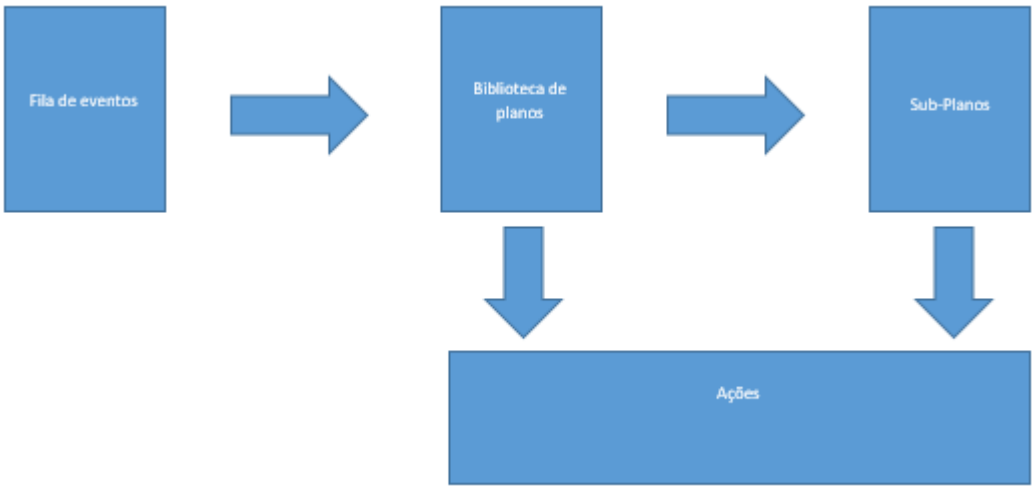

Figura 21- Ciclo do interpretador proposto
Fonte: AUTOR

Na Figura 21 não há menção do ambiente, isso ocorre porque, diferente interpretador analisados até aqui, o modelo proposto inicia-se quando já existe uma fila de eventos, sendo o interpretador apenas responsável pela seleção do evento da fila e não pela construção da fila nem da classificação dos eventos. Assim se comparado com o modelo do interpretador Jason, é possível assumir que o interpretador proposto inicia-se na etapa 5 do interpretador Jason.

Para a implementação da biblioteca de planos escolheu-se utilizar um modelo de plano parecido com o implementado pela linguagem AgentSpeak(L) e apresentado no capítulo 2.2.1. Relembrando que a estrutura de um plano em AgentSpeak(L) é formado por basicamente 3 elementos:

- Evento Trigger: evento necessário para execução.
- Contexto: pré-condições para que o plano seja executado.
- Corpo: o que o plano realmente irá realizar, seja uma ação ou subplano.

Nos cenários estudados até aqui, tanto no interpretador Jason, como no modelo PRS, uma vez escolhido um evento na fila, o interpretador deve escolher os planos que possuem como *trigger* o evento escolhido e entre esses planos é necessário escolher os que possuem um contexto satisfatório. Após isso, caso exista mais de um plano pré-selecionado, é necessário realizar algum tipo priorização para que apenas um plano seja escolhido. Trazendo esse modelo para o cenário das limitações de um sistema embarcado, faz-se mister realizar

essa etapa do interpretador utilizando o mínimo de memória RAM possível. Assim, caso se assuma que a biblioteca de planos é estática, como nos modelos visto até aqui, se percebe, analogamente, que um plano pode ser criado com a seguinte estrutura:

CASO = EventoTriger e SE contexto = verdadeiro, ENTÃO FAZ Corpo

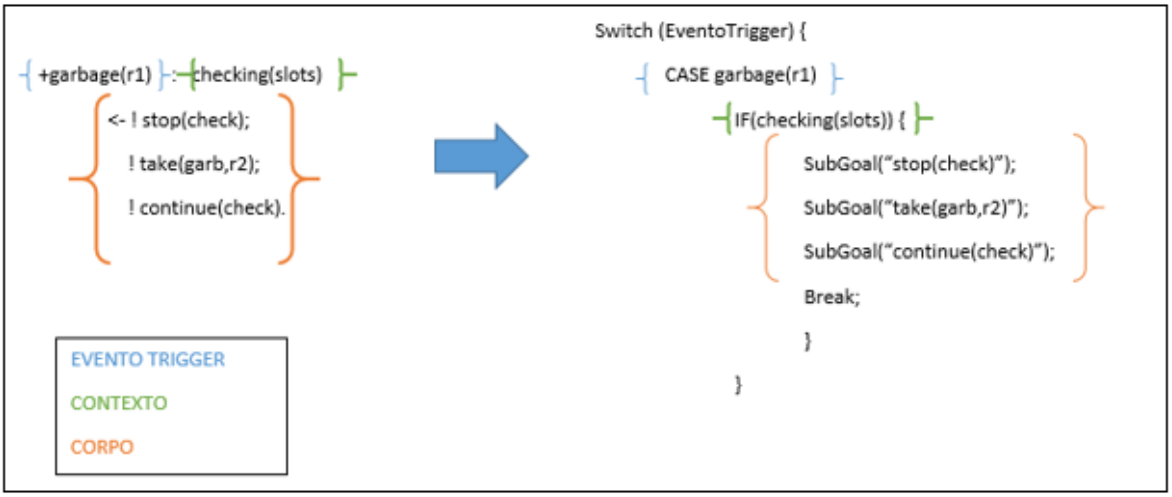

Figura 22 - Conversão de um plano
Fonte: AUTOR

A Figura 22 demonstra como é possível a conversão de um plano escrito em AgentSpeak(L), uma linguagem puramente BDI, como visto no capítulo 2.2.1, em uma estrutura de uma linguagem clássica como a linguagem C. Essa conversão se faz necessário devido a AgentSpeak(L) pertencer a 5ª geração das linguagens computacionais e os sistemas embarcados não a suportarem (NOERGAARD, 2012). A parte esquerda da Figura é um plano escrito em AgentSpeak(L) adaptado de Hübner, Bordini e Vieira (2004). A parte direita da Figura é sua tradução para a linguagem C. Não se vai ater nesse momento no corpo do plano, mas sim elementos que compõem o plano. Ambas codificações possuem os mesmos elementos. Sendo considerado isso correto, é possível escrever uma biblioteca de planos utilizando a estrutura seletiva SWITCH no caso da linguagem C, ainda mais que a estrutura permite realizar uma priorização dos planos, apenas utilizando uma ordem de codificação, sendo que planos escritos primeiros serão primeiros avaliados e terão precedência em relação aos demais. A estrutura mencionada torna-se uma solução simples, mas não simplória para a

escrita da biblioteca de planos, uma vez que consegue atender os mesmos requisitos vistos nos interpretadores estudados até aqui.

Uma vez selecionado o plano, o que deve ser feito é a execução do corpo do plano, sendo que o corpo pode possuir sub-planos ou ações. Sub-planos seguem a mesma lógica dos planos, mas já as ações são um pouco diferentes. Na construção do *framework*, optou-se por permitir que ações possam ocorrer de forma síncrona ou assíncrona. A forma síncrona é quando o corpo do plano chama uma função de ação diretamente, como por exemplo, tocar um sirene. Já na forma assíncrona, no corpo do plano pode-se apenas incluir uma crença para que a ação seja realizada pela tarefa responsável pelas ações. Esta tarefa será explicada mais nos capítulos a seguir. O *framework* permite funções assíncronas, porém se deve ter muito cuidado com esse tipo de ação, pois pode interferir diretamente na deliberação do agente, uma vez que uma ação que deveria ter ocorrido pode ainda estar pendente, esperando o escalonador do FreeRTOS, passar novamente pela tarefa de ação.

4.2.1 Armazenamento de dados

Pelo funcionamento visto até aqui o interpretador ainda não é funcional, pois o interpretador apenas está selecionando um evento na fila e executando um plano de sua biblioteca. Outro ponto central de interpretador é o acesso a base de crenças do agentes, ou seja, a quarta camada do *framework,* pois os planos são essencialmente meios para que um agente faça uma crença se tornar realizada e, por usa vez, as crenças fazem parte do contexto dos planos. Diferente da biblioteca de planos, a biblioteca de crenças não pode ser estática, pois as crenças do agente podem se alternar a todo instante. Isso leva de volta a problemática da pouca memória dos sistemas embarcados, e ainda se necessita que seja uma memória não volátil, uma EEPROM ou algo do tipo, para garantir que se houver falta de alimentação elétrica no sistema as crenças obtidas até o momento pelo agente não se percam. Sendo assim, a estratégia adotada para o *framework* foi a utilização de endereços para as crenças.

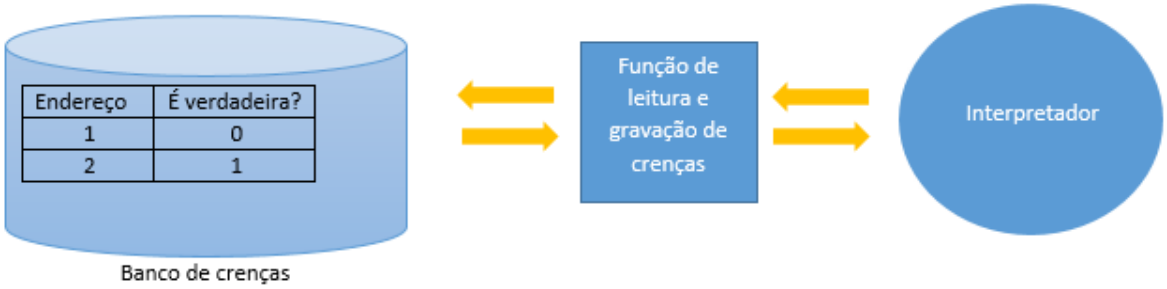

Figura 23 - Acesso a biblioteca de crenças
Fonte: AUTOR

Na Figura 23 é possível ver a forma em que o interpretador acessa a base de crenças, basicamente toda vez que é necessário realizar uma consulta, gravação ou atualização de crença, o interpretador invoca a função responsável por isso. A função por sua vez faz o encapsulamento da chamada de baixo nível. Assim fica invisível para o interpretador onde realmente está a base de crenças, sendo que a base pode estar em uma memória EEPROM interna ou externa, em um cartão de memória SD, ou até mesmo em um servidor *web* externo ao agente. Com essa estratégia de armazenamento da base de crenças o *framework* proposto gera ao programador uma flexibilidade imensa, uma vez que não deve haver limite para o tamanho da base de crenças, pois não utilizará a memória limitada do sistema. Contudo, vale ressaltar que essa estratégia gera um novo inconveniente, que é a latência das memórias externas, sendo que essas geralmente possuem um tempo de escrita e leitura mais lenta, como visto no capítulo 1.1.2. Assim é necessário ter um certo cuidado na escolha do tipo de armazenamento que será empregado para guardar a base de crenças. Detalhes sobre o código fonte das funcionalidades mencionadas até aqui serão vistos no próximo capítulo, de forma que não se detenha em especificações técnicas nesse momento.

Observa-se que o interpretador proposto em nenhum momento fez acesso as camadas mais baixas do sistema, ou seja, não houve interação com bibliotecas de *hardware*. Contudo foi visto que o interpretador necessita de dados do ambiente. Para entender como *framework* proposto coleta dados do ambiente, é necessário entender como estão estruturadas as tarefas ou *TASKs* do RTOS, dentro do *framework* proposto.

4.2.2 Tarefas

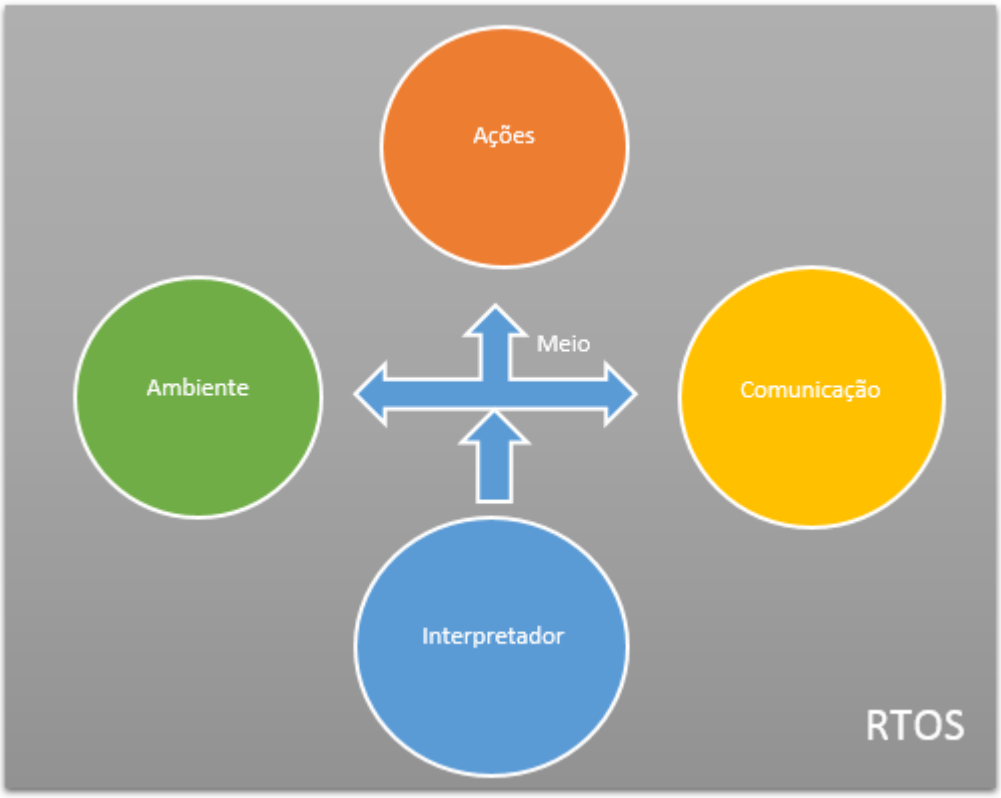

Figura 24 - Tarefas dentro do RTOS
Fonte: AUTOR

Na Figura 24 é possível ver as 4 tarefas que estão dentro do FreeRTOS. Como visto no capítulo 4.1 a segunda camada do *framework* é composto pelo RTOS e a terceira camada é o interpretador. No entanto, apesar da terceira camada ter como principal elemento o interpretador, essa camada possui mais 3 elementos.

- Ambiente: controla leituras de sensores de ambiente.
- Ações: controla os atuadores do sistema.
- Comunicação: controla a comunicação do sistema.

Esses quatros elementos se tratam de tarefas em execução dentro do escalonador do FreeRTOS. Como mencionado, o interpretador proposto se comparado com o interpretador do Jason, contempla as etapas de 5 a 10 do Jason. Da mesma forma é possível afirmar que a tarefa de ambiente contempla as tarefas 1 e 2; e as etapas 3 e 4 são realizadas pela tarefa de comunicação. Assim é possível realizar uma comparação da camada 3 do *framework* com o interpretador Jason, como pode ser visto no Quadro 3

Quadro 3 – Comparação de interpretadores
Fonte: AUTOR

Etapas do interpretador Jason	Tarefa correspondente
1	Tarefa de ambiente
2	Tarefa de ambiente
3	Tarefa de comunicação
4	Tarefa de comunicação
5	Tarefa interpretador
6	Tarefa interpretador
7	Tarefa interpretador
8	Tarefa interpretador
9	Tarefa interpretador
10	Tarefa interpretador

No próximo capítulo será explicado o funcionamento prático de cada uma dessas tarefas.

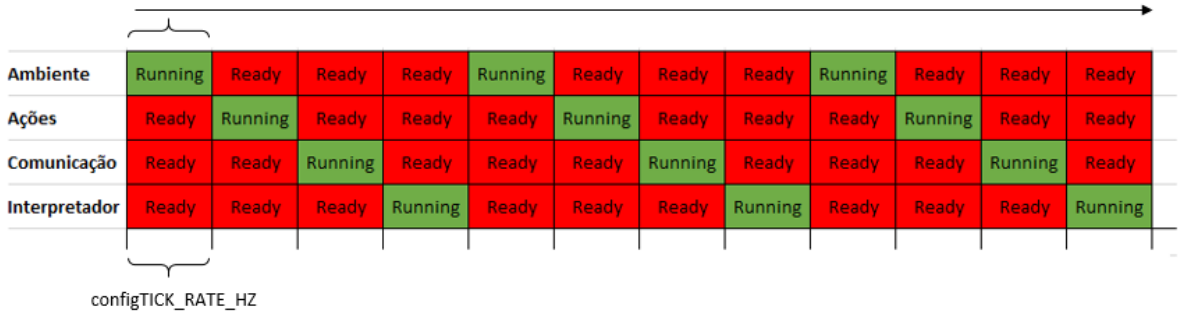

Figura 25 – Troca de contextos entre tarefas
Fonte: AUTOR

A Figura 25, mostra como ocorre a alternância dos estados entre as tarefas. Conforme visto no capítulo 5.1, uma tarefa no FreeRTOS pode variar em 4 estados, porém no caso do *framework* proposto, apenas 2 estados são utilizados, '*Ready*' e '*Running*'. Durante a execução o escalonador do FreeRTOS apenas deixa um tarefa em execução. A execução dura o número de *hertz* configurados pelo parâmetro 'configTICK_RATE_HZ'. Essa configuração é realizada no arquivo 'FreeRTOSConfig.h', como já explicado. O que deve ser visto aqui é que a alternância das tarefas ocorre devido a um número de ciclos, portanto um sistema que possuí um oscilador de maior frequência, possui uma alternância mais rápida em relação ao

mesmo tempo de um oscilador mais lento. Não são apresentados detalhes aqui sobre osciladores em microcontroladores, mas basicamente todo microcontrolador necessita de um oscilador para funcionar e a frequência suportada por cada microcontrolador é definido pelo fabricante. Outro ponto importante sobre a duração de execução de cada tarefa, é o fato que ela não ocorre de forma instantânea, mesmo que isso não seja percebido no funcionamento. Quando uma tarefa é alterada de estado, há um consumo de recurso por parte do RTOS, assim deve-se ter em mente que se o ciclo de alternância entre tarefas for muito curta, há desperdício de processamento, pois o processador tende a ficar mais tempo processando a troca entre estados das tarefas que processando as funções das tarefas em si. A troca entre tarefas em execução é chamada troca de contexto. Esse processo tende a utilizar um volume alto de memória SRAM, pois a execução da tarefa é pausada e seu estado salvo antes da tarefa ser colocada no estado de '*Ready*'. Com isso é possível dizer que quanto mais tarefas em espera, maior é o consumo de memória. Com essa problemática em mente, no momento da construção do *framework* foram criadas apenas as tarefas essenciais para o funcionamento, visando uma execução mais otimizada possível em termos de consumo de memória.

4.3 Implementação

Para o desenvolvimento da proposta, foi escolhida a utilização da IDE Eclipse utilizando o plug-in AvrDude, não entra-se aqui em detalhes sobre a IDE nem sobre o *plugin*. Todo o código fonte está disponível no GitHub do *framework* proposto (ERADE).

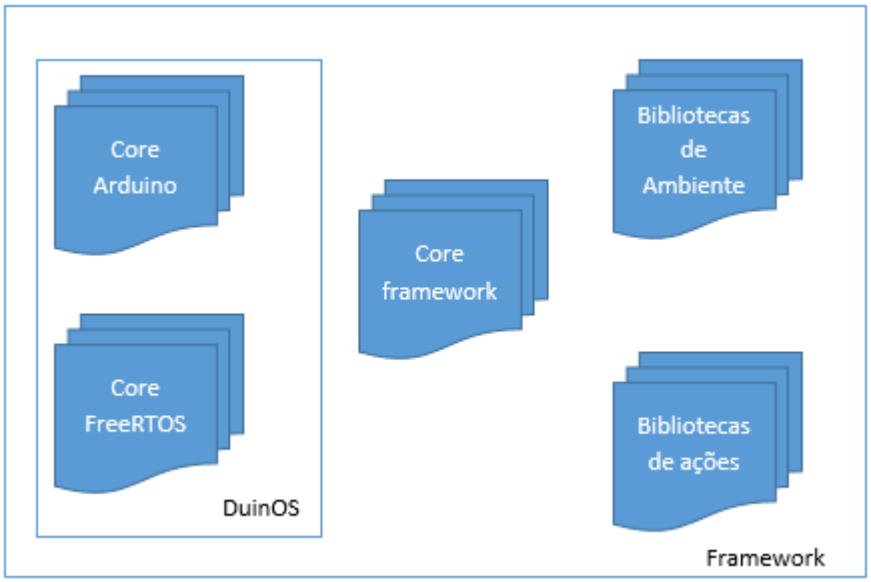

Figura 26- Estrutura dos códigos
Fonte: AUTOR

A Figura 26 mostra como é a organização do código, pode-se ver que dentro do *framework* existem 5 grandes grupos, dentro desses 5 grupos tem-se o subgrupo DuiOS que é formado pelos arquivos do FreeRTOS e do Arduino.

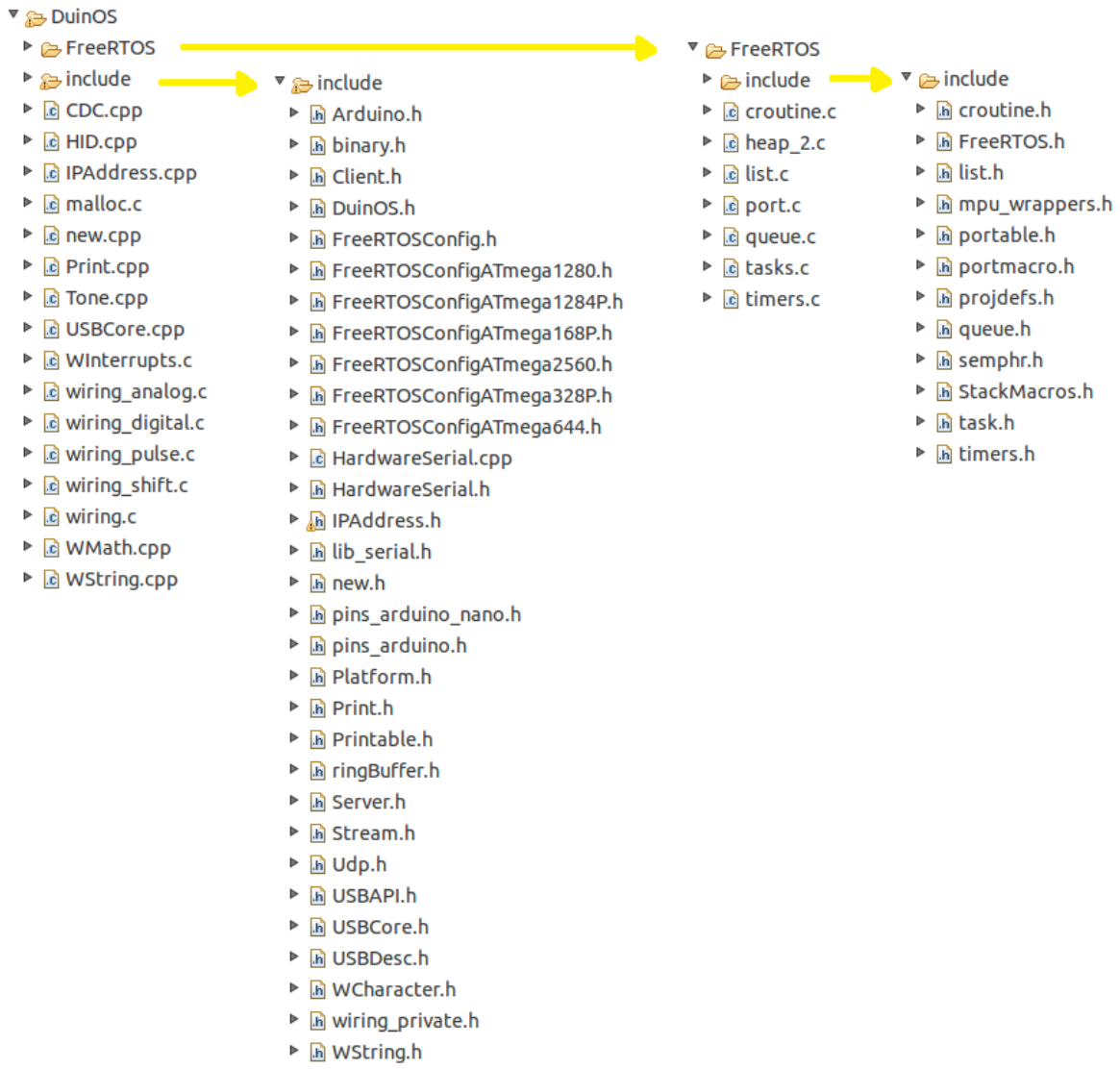

Figura 27 - Arquivos DuinOS
Fonte: AUTOR

Como visto anteriormente o DuinOs basicamente é uma junção de arquivos do Arduino com os do FreeRtos. Como dito também, no *framework* proposto é usado apenas o *core* do DuinOS, pois a plataforma de desenvolvimento escolhida foi o Eclipse e não a IDE Arduino, isso fez com que o restante do projeto DuinOS fosse descartada. A Figura 27

demonstra todos os arquivos utilizados do DuinOS. Como padrão no projeto do *framework*, todo arquivo *Header*(.h), está localizado dentro de uma pasta chamada 'Include'. Nota-se que existe uma pasta 'include' dentro da raiz da pasta DuinOS e uma dentro da pasta FreeRTOS. Os arquivos da raiz do DuinOS são todos arquivos do Arduino, assim a pasta 'include' da raiz são os arquivos 'header' do Arduino. Da mesma forma, o conteúdo da pasta FreeRTOS, possui os arquivos do core do FreeRTOS e a pasta include os arquivos 'header' do FreeRTOS. Um detalhe importante é que o arquivo 'Main.cpp', não está na Figura 27. O arquivo foi removido de forma proposital, pois como será visto mais à frente o *framework* possui um arquivo 'main' diferente.

E necessário ressaltar os 3 principais arquivos desse grupo, sendo eles:

- FreeRTOSConfig.h: como visto no capítulo 5.1, esse arquivo é responsável por realizar a configuração de execução do FreeRTOS, porém neste caso, ele apenas serve como um desvio para outro arquivo *header*. Como pode ser visto na Figura 28, quando é invocado o arquivo FreeRTOSConfig.h, ele automaticamente desvia o chamado para o arquivo de configuração do microcontrolador que está configurado no AvrDude. A configuração do AvrDude, sempre cria arquivos de DEFINE, assim é possível saber qual microcontrolador foi configurado apenas utilizando a função 'defined()'.

```
#if defined(__AVR_ATmega644__) || defined(__AVR_ATmega644P__)
#include "FreeRTOSConfigATmega644.h"

#elif defined(__AVR_ATmega1284P__)
#include "FreeRTOSConfigATmega1284P.h"

#elif defined(__AVR_ATmega1280__)
#include "FreeRTOSConfigATmega1280.h"

#elif defined(__AVR_ATmega2560__)
#include "FreeRTOSConfigATmega2560.h"

#elif defined(__AVR_ATmega328P__)
#include "FreeRTOSConfigATmega328P.h"

#elif defined(__AVR_ATmega88__) || defined(__AVR_ATmega88P__) || defined(__AVR_ATmega168__) || defined(__AVR_ATmega168P__)
#include "FreeRTOSConfigATmega168P.h"
#else
#error "Device is not supported by DuinOS"

#endif
```

Figura 28 –Arquivo FreeRTOSConfig.h
Fonte: AUTOR

- FreeRTOSConfgAtMegaXXX.h: esse arquivo por sua vez, é o arquivo que realmente possui a configuração do FreeRTOS, ou seja, esse é o arquivo para qual o FreeRTOSConfig.h desvia sua chamada. Na Figura 29 é possível ver o exemplo de uma configuração para o microcontrolador AtMega2560. Para cada microcontrolador

deve-se ter um arquivo de configuração, pois as necessidades e limitações são diferentes. É possível ver que no exemplo mostrado o tamanho máximo de memória 'Heap' está configurado para 4096 bytes, isso só é possível pois esse microcontrolador possui 8 KBts de SRAM. Caso fosse utilizado um microcontrolador como o AtMega328p, essa configuração seria totalmente errada, pois o microcontrolador possui apenas 1 KBts de SRAM.

```
#define configMINIMAL_STACK_SIZE   ( ( unsigned portSHORT ) 85 )
#define configTOTAL_HEAP_SIZE      ( (size_t ) ( 4096 ) )
#define configMAX_TASK_NAME_LEN    ( 16 )
#define configUSE_TRACE_FACILITY   0
#define configUSE_16_BIT_TICKS     1
#define configIDLE_SHOULD_YIELD    0
#define configQUEUE_REGISTRY_SIZE  0

/* Co-routine definitions. */
//##2009.10.20: defined as "0":
#define configUSE_CO_ROUTINES      0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* setup() and loop() parameters */
#define configSETUP_STACK_SIZE          (configMINIMAL_STACK_SIZE * 4)
#define configLOOP_STACK_SIZE           (configMINIMAL_STACK_SIZE * 2)
#define configSETUP_PRIORITY           HIGH_PRIORITY
#define configLOOP_PRIORITY            LOW_PRIORITY

/* Set the following definitions to 1 to include the API function, or zero
 to exclude the API function. */
#define INCLUDE_vTaskPrioritySet       1
#define INCLUDE_uxTaskPriorityGet      1
//##If the following value is set to 1, change the memory managment scheme to heap_2.c:
#define INCLUDE_vTaskDelete            1
#define INCLUDE_vTaskCleanUpResources  0
#define INCLUDE_vTaskSuspend           1
```

Figura 29 – Arquivo FreeRTOSConfgAtMegaXXX.h
Fonte: AUTOR

- Pins_Arduino.h: parecido com o arquivo anterior, o arquivo Pins_Arduino.h também serve para que o sistema operacional possa funcionar em vários microcontroladores. Porém diferente do arquivo anterior esse arquivo não configura limitações de memória por exemplo, e sim ele faz um mapeamento dos pinos que o microntrolador possui, pois cada microcontrolador possui pinos diferentes e tipos de saídas diferentes. Para mais detalhes sobre a pinagem dos microcontroladores é necessário verificar o *datasheet* do modelo.

Completando os 5 grupos de arquivos, temos os arquivos do *core* do *framework*, as bibliotecas de ambiente e de ações. A Figura 30, mostra a estruturação desses 3 grupos de arquivos.

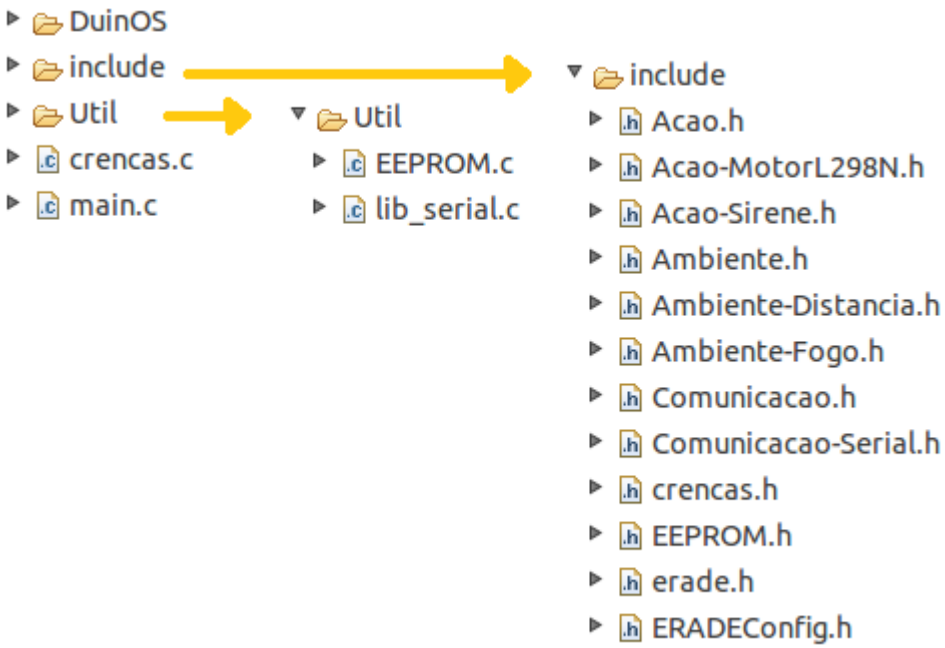

Figura 30 – Arquivos do *framework* proposto
Fonte: AUTOR

Como dito anteriormente todo arquivo '.h' deve ficar dentro de uma pasta 'include', por isso é possível ver a pasta 'include' na Figura 30. Outra convenção utilizada, foi a inclusão no nome do arquivo a sua funcionalidade, como por exemplo, todo o arquivo de ação, terá em seu nome 'acao', isso serve para bibliotecas de ambiente e de comunicação. Todos os outros arquivos são considerados arquivos de núcleo.

Claramente o arquivo mais importante do *framework* é o arquivo 'main'. Este arquivo é o responsável por iniciar o sistema e as tarefas do FreeRTOS. Neste também estão o código das tarefas em si. É possível dividi-lo em 3 partes, a parte de criação das tarefas, o *setup* de inicialização e o código das tarefas.

A Figura 31 mostra o código referente a criação das tarefas e a sua inicialização, não se faz necessário a explicação desse trecho, uma vez que o capítulo 5.1 explica como é realizada a criação de uma tarefa no FreeRTOS. Apenas é necessário ressaltar, que como mostra a Figura 31, todas tarefas tem a mesma prioridade, neste caso 3. Assim nenhuma das tarefas terá prioridade na execução. Outro ponto importante, é que como pode ser visto o parâmetro 'usStackDepth' varia em cada tarefa. Mostrando valores maiores para as tarefas de ambiente e comunicação.

```
static void interpretador(void *pvParameters);
static void ambiente(void *pvParameters);
static void comunicacao(void *pvParameters);
static void acao(void *pvParameters);

/* Main program loop */
int main(void) __attribute__ ((OS_main));
                    :
int main(void)      . . .
                    :
{
    setup();

    xTaskCreate(
        interpretador
        ,  (const portCHAR *)PSTR("interpretador")
        ,  200
        ,  NULL
        ,  3
        ,  NULL );

    xTaskCreate(
        ambiente
        ,  (const portCHAR *)"Ambiente"
        ,  300
        ,  NULL
        ,  3
        ,  NULL );

    xTaskCreate(
        comunicacao
        ,  (const portCHAR *)"Comunicacao"
        ,  300
        ,  NULL
        ,  3
        ,  NULL );

    xTaskCreate(
        acao
        ,  (const portCHAR *)PSTR("Acao")
        ,  100
        ,  NULL
        ,  3
        ,  NULL );
    vTaskStartScheduler();

    #ifdef _DEBUG
        enviar( PSTR("\r\n\n\nFalta de espaço!!\r\n"));
    #endif
}
```

Figura 31 – Arquivo Main - parte 1
Fonte: AUTOR

Na Figura 32 é possivel ver a função ‘setup’ do sistema. Na versão atual do *framework*, essa etapa apenas serve para limpar os dados da EEPROM e iniciar uma porta serial que pode ser utilizada em qualquer ponto do sistema. Em versões futuras essa função poderá ser utilizada para realizar outras configuraçãoes do sistema, sendo essa função executada uma vez, apenas quando o sistema é iniciado. Se faz necessário notar que em vários pontos do sistema existe a expressão _DEBUG. Essa contante foi criada para habilitar alguns avisos caso o sistema esteja em modo de depuração.

```
void setup()
{
    for (int i = 0 ; i < 1024 ; i++) {
       gravar(i, 0);
     }

    xSerialPort = xSerialPortInitMinimal( USART0, 9600, portSERIAL_BUFFER_TX, portSERIAL_BUFFER_RX);

   #ifdef _DEBUG
        avrSerialxPrint_P(&xSerialPort, PSTR("\r\n\n\nIniciando sistema\r\n"));
   #endif
}
```

Figura 32 – Arquivo Main - parte 2
Fonte: AUTOR

A terceira parte do arquivo Main.c é o corpo das 4 tarefas mencionadas no capítulo anterior, sendo elas: a tarefa de ambiente, ação, comunicação e interpretador.

```
static void ambiente(void *pvParameters){

    carregarAmbiente();

    for(;;){
        #ifdef E_DISTANCIA
            AmbienteDistanciaRun();
        #endif
        #ifdef E_FOGO
            AmbienteFogoRun();
        #endif

        #ifdef _DEBUG
            delay(1000);
        #endif
    }
}

static void acao(void *pvParameters)
{

    carregarAcoes();

    for (;;) {

        #ifdef _DEBUG
            enviar(PSTR("\n\n\n\n task acao \n\n"));
            delay(1000);
        #endif
    }
}

static void comunicacao(void *pvParameters)
{
    for (;;) {

        int8_t recebido = receber();
        gravar(C_RECEBIDO, recebido );

        #ifdef _DEBUG
            avrSerialxPrintf_P(&xSerialPort, PSTR("\n\nRecebido ---> %u\r\n"), recebido  );
            enviar(PSTR("\n\n\n\n task comunicacao \n\n"));
            delay(1000);
        #endif
    }
}

static void interpretador(void *pvParameters)
{
    for(;;){

        CRENCA crenca;
        crenca = (int) ler(UltimaCrenca);

        planos(crenca);

        #ifdef _DEBUG
            avrSerialxPrintf_P(&xSerialPort, PSTR("Memoria Heap livre: %u\r\n"), xPortGetFreeHeapSize() );
            delay(200);
        #endif

    }
}
```

Figura 33 – Arquivo Main - parte 3
Fonte: AUTOR

Como visto na Figura 33, todas as tarefas são funções com um loop infinito, utilizando o conceito de super-loop, como já mencionado no capítulo 5.1. A seguir explica-se com mais detalhes o código de cada uma das tarefas.

4.3.1 Ambiente

A tarefa do ambiente controla os sensores do sistema. A Figura 34 representa o código dessa tarefa.

```
static void ambiente(void *pvParameters){

    carregarAmbiente();

    for(;;){
        #ifdef E_DISTANCIA
            AmbienteDistanciaRun();
        #endif
        #ifdef E_FOGO
            AmbienteFogoRun();
        #endif

        #ifdef _DEBUG
            delay(1000);
        #endif
    }
}
```

Figura 34 – Arquivo Main - controle de ambiente
Fonte: AUTOR

É necessário reparar que a chamada da função 'carregarAmbiente' não é realizada dentro do loop, isso porque essa função tem o objetivo de fazer o *setup* dos sensores e deve ser chamada apenas na inicialização do sistema. Dentro do loop, pode-se ver que existem consultas a alguns DEFINEs. Essa técnica foi utilizada para que o sistema verifique que sensor está habilitado e execute o método '*run*' de cada um. Assim todo o sensor que será utilizado deve estar dentro desse *loop*. No exemplo da Figura 34, podemos ver que só existem 2 sensores nessa fase do *framework*. Por convenção utilizou a letra E como antecessora dos DEFINEs de sensor. Esses DEFINEs devem estar codificados no arquivo ERADEConfig.h. No próximo capítulo, quando será apresentado um projeto piloto realizado com o *framework* essa configuração ficará clara.

Figura 35 – Fluxo de controle do ambiente
Fonte: AUTOR

A Figura 35 representa o ciclo da tarefa de controle do ambiente. O ciclo se inicia verificando os sensores que estão habilitados, isso ocorre com a chamada da função ‘carregarAmbiente’. Logo após, já dentro do *loop*, o método de execução de cada sensor é invocado. Já a última etapa ocorre dentro de cada biblioteca de sensor. Para exemplificar, utilizara-se aqui a biblioteca ‘Ambiente-Fogo.h’.

Essa biblioteca foi escrita pelo autor com o objetivo de comunicar com um sensor de chama, que nada mais consiste que um sensor infravermelho capaz de detectar chamas em um ambiente. A Figura 36 mostra o código da biblioteca mencionada.

```
#ifndef AMBIENTE_FOGO_H
#define AMBIENTE_FOGO_H

#include "Ambiente.h"

#define PinFire 28

void AmbienteFogoSetup(void) {
      pinMode(PinFire,INPUT);
}

void AmbienteFogoRun(void) {
        #ifdef _DEBUG
            avrSerialxPrint_P(&xSerialPort, PSTR("\r\n\n\nTask Fogo\r\n"));
        #endif

        int sensorReading = digitalRead(PinFire);

          if (sensorReading==1) {
              retirarCrenca(E_FOGO);
          }

         else {

            #ifdef _DEBUG
              avrSerialxPrint_P(&xSerialPort, PSTR("\r\n\n\n*fogo*\r\n"));
            #endif

            //gravar
            incluirCrenca(E_FOGO);
          }
};

#endif
```

Figura 36 – Biblioteca Ambiente-fogo.h
Fonte: AUTOR

É possível perceber que essa biblioteca é bem simples, por isso não entra-se aqui em detalhes técnicos sobre as funções secundarias. Essa biblioteca segue o padrão do *framework* para bibliotecas de ambiente, possuindo apenas 2 funções, uma para *setup* e outra para *run*. A função *run* verifica se o sensor está em 1 ou 0, caso esteja em 0 é porque existe uma chama, caso contrário não existe chama detectada. É possível notar também a chamada de duas funções, 'incluirCrenca' e 'retirarCrenca', uma inclui a crença que existe fogo e outra a retira, respectivamente. Essa inclusão é realizada através de endereços, como explicado no capítulo anterior.

4.3.2 Ações

A função da tarefa de controle de ações é muito parecida com a do ambiente. A Figura 37 mostra o código dessa tarefa.

```
static void acao(void *pvParameters)
{

    carregarAcoes();

    for (;;) {

        #ifdef _DEBUG
            enviar(PSTR("\n\n\n\n task acao \n\n"));
            delay(1000);
        #endif
    }
}
```

Figura 37 – Arquivo Main - controle de ações
Fonte: AUTOR

Essa tarefa também possui a chamada de uma função para o *setup* dos atuadores, porém no *loop*, como pode ser visto, não possui nenhuma chamada. Isso porque neste exemplo não existe nenhuma ação assíncrona configurada. Como dito no capítulo anterior é possível incluir uma crença para que seja executada uma função. Para fazer isso, apenas se faz necessário colocar a chamada da função 'ConsultarCrenca()', passando o nome da crença que deve ser verificada e em seguida a função de ação que deve chamada. O *framework* permite esse tipo de ação, porém as chamadas de ações síncrona se mostram muito mais eficiente.

4.3.3 Comunicação

A comunicação na versão atual do *framework* está dividida em apenas duas funções, uma de envio e uma de recebimento. Como visto na Figura 38, a função de recebimento é realizado durante o *loop*, isso faz com que cada ciclo do escalonador um tempo seja reservada para o recebimento de mensagens. Uma vez recebida a mensagem ela é armazenada no endereço C_RECEBIDO. O envio de mensagens ocorre através, da chama da função 'enviar'. Essa função é utilizada com uma chamada de forma direta. A biblioteca que será utilizada para fazer a comunicação, deve possuir esses 2 métodos e deve ser configurada no arquivo 'comunicacao.h'. Como foi visto no capítulo 2.3, a comunicação do agente é complexa, pois existem outros fatores além da simples comunicação, como por exemplo a aceitação social, por isso na versão atual do *framework*, se optou por uma comunicação simples, deixando para trabalhos futuros a reconstrução da estrutura de comunicação do *framework*.

```
static void comunicacao(void *pvParameters)
{
    for (;;) {

        int8_t recebido = receber();
        gravar(C_RECEBIDO, recebido );

        #ifdef _DEBUG
            avrSerialxPrintf_P(&xSerialPort, PSTR("\n\nRecebido ---> %u\r\n"), recebido  );
            enviar(PSTR("\n\n\n\n task comunicacao \n\n"));
            delay(1000);
        #endif
    }
}
```

Figura 38 – Arquivo Main - controle de comunicação
Fonte: AUTOR

4.3.4 Interpretador

Como visto no capítulo anterior, o interpretador tem como função principal selecionar um evento ocorrido e escolher um plano para a execução. A tarefa do interpretador então consiste, neste momento, em buscar o último evento ocorrido, ou seja, a última crença que se tornou verdade e submeter a função chamada 'planos'. Essa função, por sua vez, é formada pela biblioteca de planos, onde estão todos os planos estruturados, como explicado no capítulo anterior. A Figura 39 mostra o código dessa tarefa.

```
static void interpretador(void *pvParameters)
{
    for(;;){

        CRENCA crenca;
        crenca = (int) ler(UltimaCreca);

        planos(crenca);

        #ifdef _DEBUG
            avrSerialxPrintf_P(&xSerialPort, PSTR("Memoria Heap livre: %u\r\n"), xPortGetFreeHeapSize() );
            delay(200);
        #endif

    }
}
```

Figura 39 – Arquivo Main - interpretador
Fonte: AUTOR

Como visto aqui, no desenvolvimento de todas as funções do *framework*, foram utilizados todos os conceitos estudados até o momento com a produção mínima de código, visando as restrições dos sistemas embarcados e evitando que o *framework* consuma acessivo espaço de armazenamento. No próximo capítulo é explicado um projeto simples utilizando o *framework* e como a proposta minimiza e simplifica um desenvolvimento de um projeto.

5 PROJETO PILOTO

5.1 Especificação do projeto

Para o projeto piloto do *framework*, utilizara-se uma proposta parecida com o famoso robô aspirador proposto por Russel e Norvig (2003). A Figura 40 demostra a ideia do projeto. Imagina-se aqui um robô cujo objetivo será verificar se um determinado ambiente possuí algum sinal de fogo, caso exista o robô deve acionar uma sirene avisando as pessoas a sua volta.

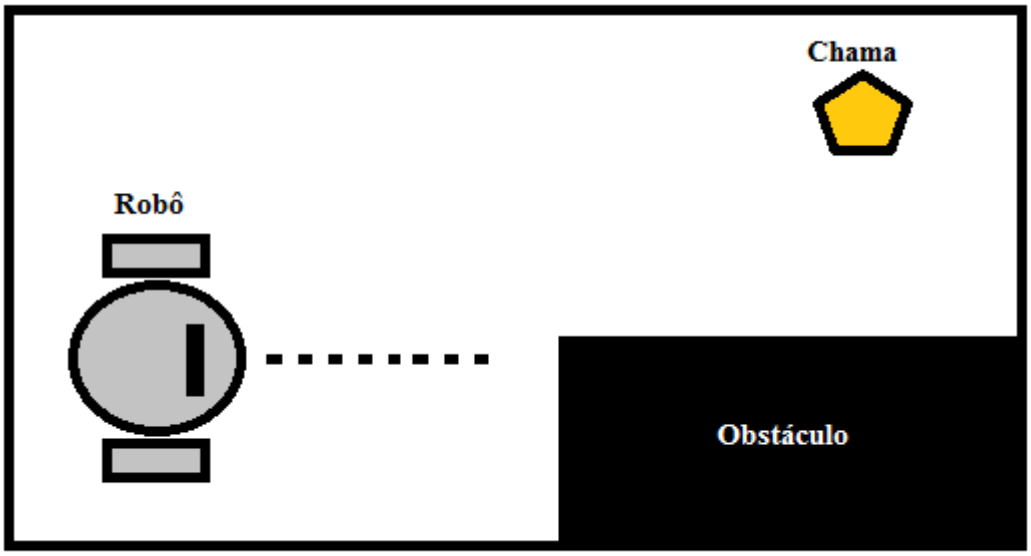

Figura 40 – Robô proposto
Fonte: AUTOR

Para realizar essa tarefa, o robô deve conseguir andar pelo ambiente e desviar de obstáculos. Não entra-se em detalhes sobre especificações técnicas de cada *hardware*. Basicamente o que é necessário, em termos de *hardware*, para realizar dessa tarefa são:

- Sensor de chamas: este sensor é capaz de verificar se há alguma chama. A opção escolhida para o projeto foi utilizar um sensor óptico para essa função. A Figura 41 mostra a aparência desse sensor.

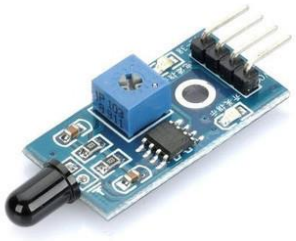

Figura 41 – Sensor de chamas
Fonte: AUTOR

- Sensor ultrassom: para que seja possível desviar de obstáculos e evitar que o robô se aproxime demais das chamas, utilizou-se um sensor de ultrassom. Este sensor é capaz de calcular a distância até um objeto. Esse cálculo é realizado pois o sensor emite um sinal ultrassonoro e capta seu retorno, ou seja, quanto mais rápido o sinal retornar mais perto do obstáculo ele está. A Figura 42 mostra o sensor utilizado.

Figura 42 – Sensor ultrassom
Fonte: AUTOR

- Motores: obviamente o robô proposto deve ser capaz de se movimentar, assim claramente se faz necessário que existam motores e rodas para a movimentação do mesmo. A Figura 43 representa os motores utilizados.

Figura 43 – Motores
Fonte: AUTOR

- Ponte H: como dito anteriormente não se entra em detalhe sobre questões de *hardware* neste projeto, assim talvez este componente possa parecer estranho para quem não está habituado com eletrônica. Se faz necessário a utilização de uma ponte H, pois é ela que fornece potência suficiente para os motores, uma vez que os microcontroladores não conseguem trabalhar com potências diferentes de microampères, valores muito aquém do necessário para movimentar a estrutura do robô proposto. A Figura 44 mostra a ponte H que foi utilizado no projeto.

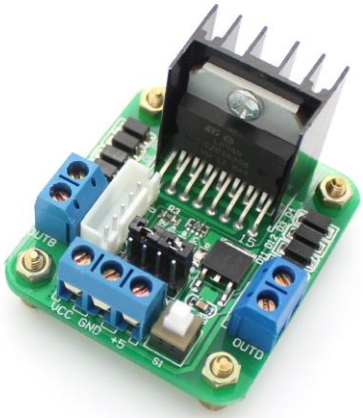

Figura 44 – Ponte H
Fonte: http://arduino.stackexchange.com/questions/3682/question-different-ways-of-connecting-l298n-motor-driver-board-to-arduino-and-mo

As Figuras 45, 46 e 47 mostram o projeto montado com todos os componentes eletrônicos e o *chassis* do robô, no próximo capítulo é visto a codificação do projeto e a utilização do *framework* neste processo.

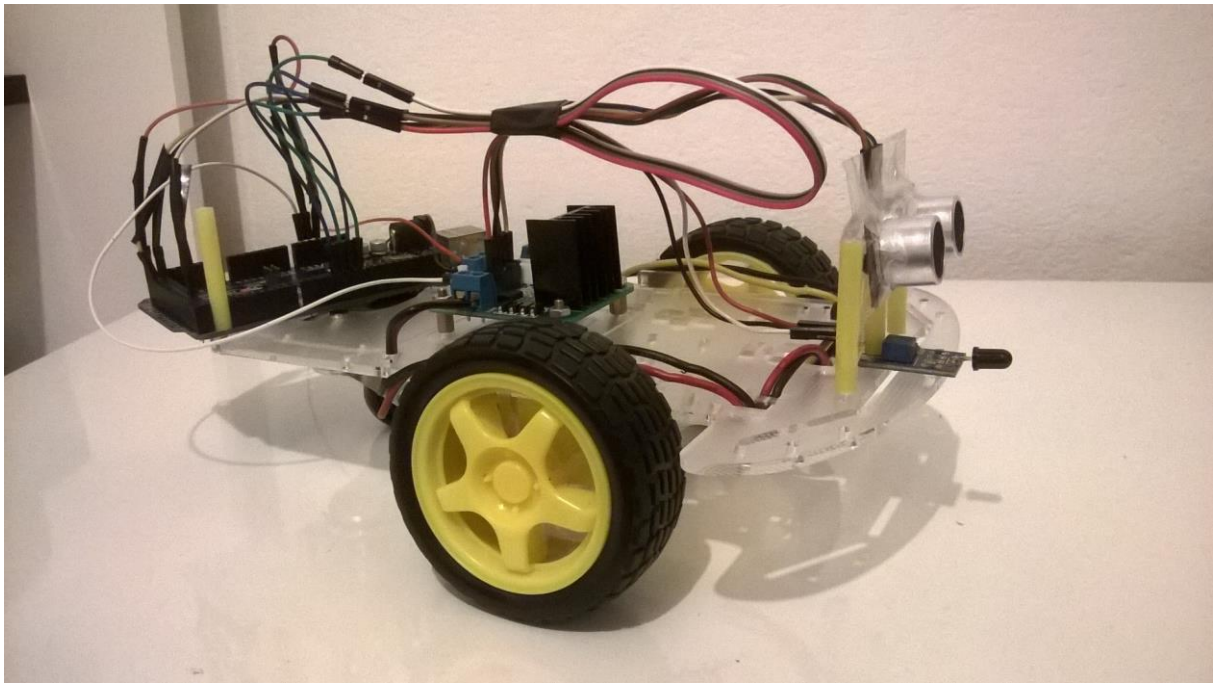

Figura 45 – Robô montado - perfil
Fonte: AUTOR

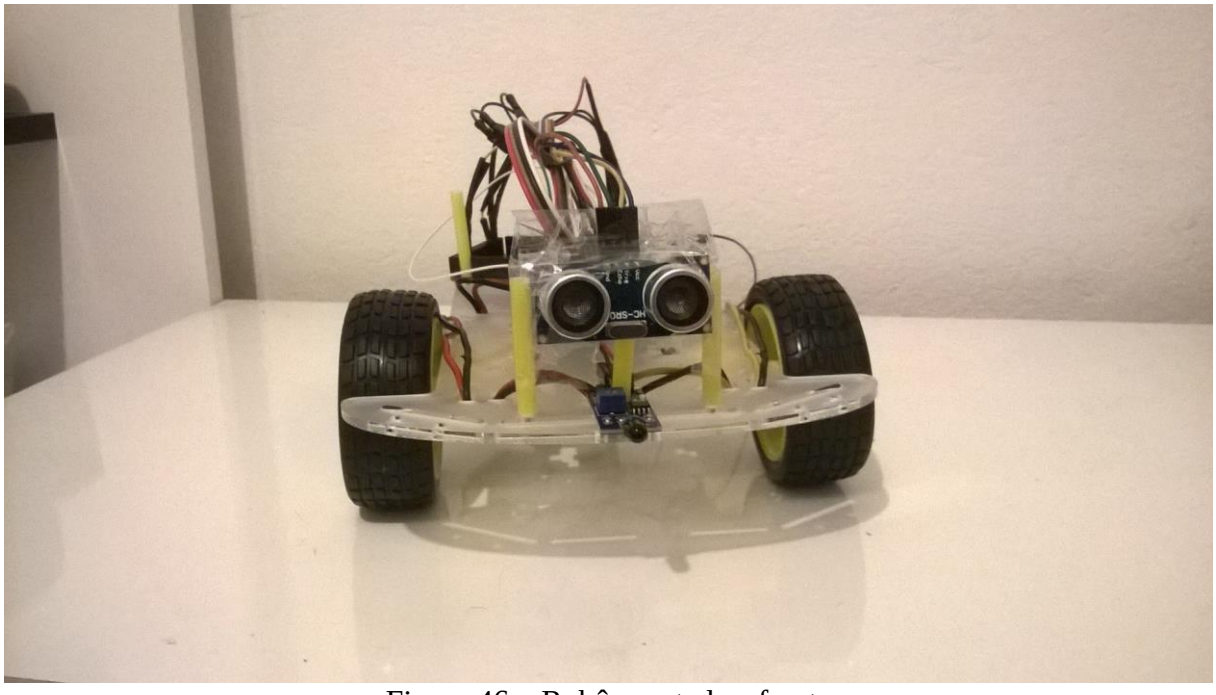

Figura 46 – Robô montado - frente
Fonte: AUTOR

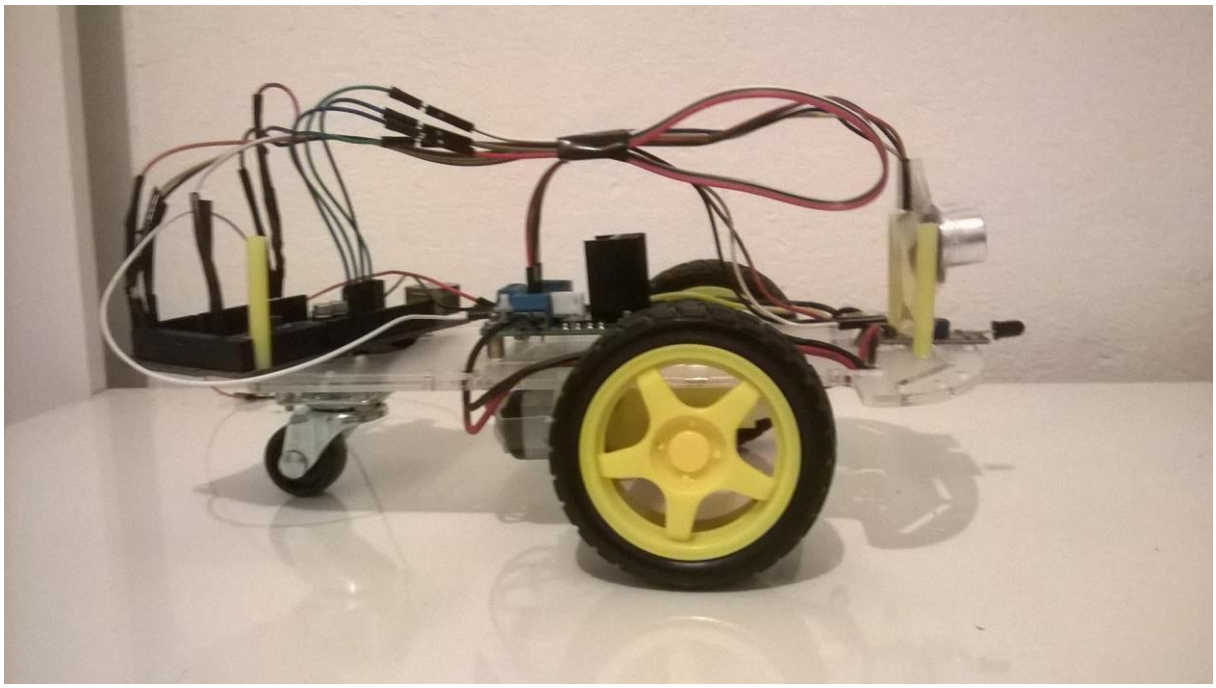

Figura 47 – Robô montado - lado
Fonte: AUTOR

5.2 Desenvolvimento

Inicia-se o desenvolvendo do projeto piloto preparando o ambiente de desenvolvimento com o *framework* proposto. Como dito anteriormente não entra-se em detalhes neste trabalho sobre a IDE utilizada, porém os próximos capítulos demonstram na prática como a ferramenta foi utilizada e logo a seguir a explanação técnica sobre o código utilizado.

5.2.1 Preparação do *framework* no ambiente

Com a IDE Eclipse aberta, o primeiro passo é realizar a importação do código do *framework*. Para realizar isso, deve-se ir em **File->Import**, como mostra a Figura 48.

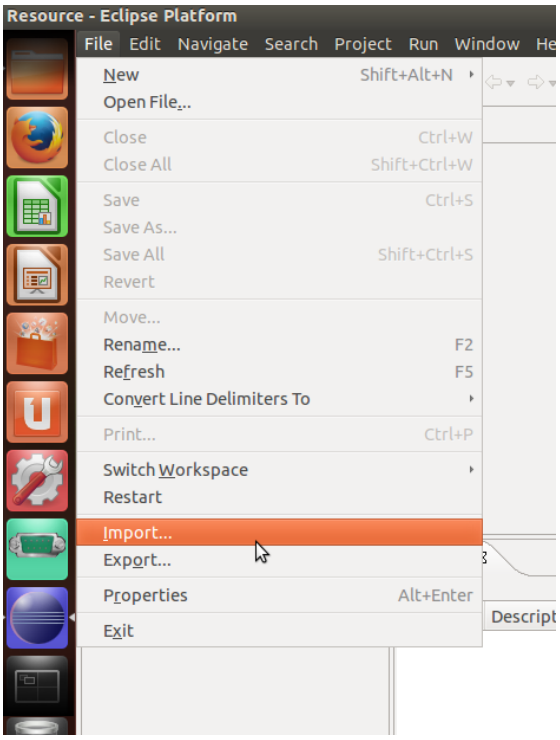

Figura 48 – Importando o *framework*
Fonte: AUTOR

A Figura 49, mostra as opções de importação, escolha a que se adéque com a forma de como foi salvo o *framework*.

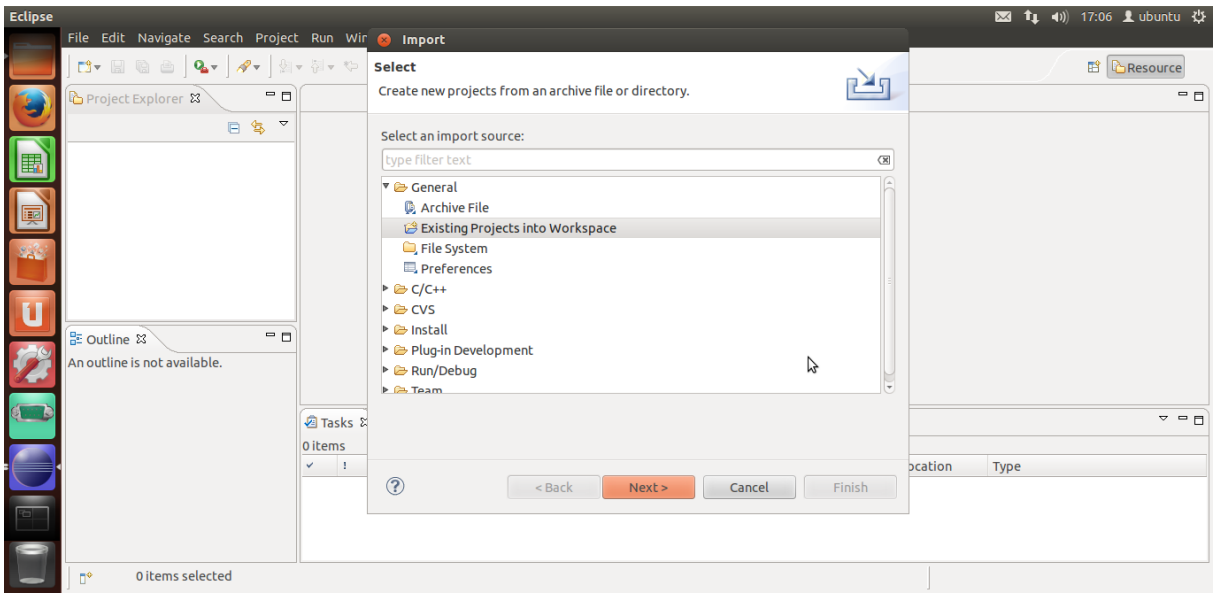

Figura 49 – Escolhendo o tipo de importação
Fonte: AUTOR

Localize-se onde está salvo o *framework* e marca-se a opção '**copy projects into workspace'.**

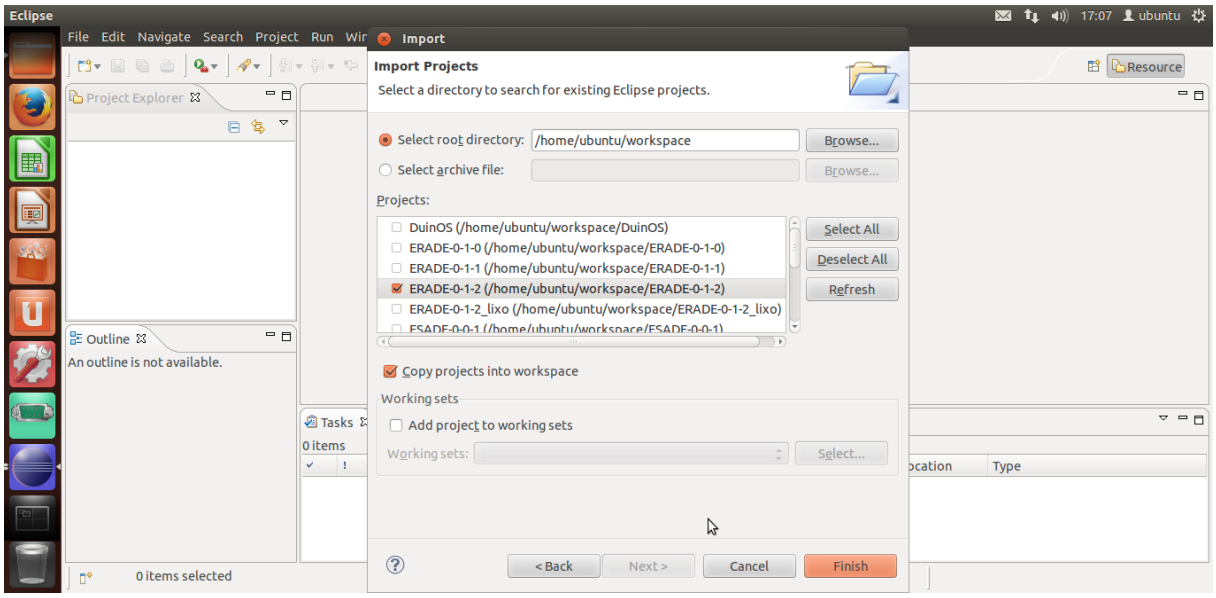

Figura 50 – Selecionando o arquivo
Fonte: AUTOR

Realizado esse passo, já se tem o projeto com o *framework* criado. Agora é necessária a configuração do projeto. Para isso deve-se ir nas propriedades do projeto, que pode ser acessado através do botão direito do mouse e selecionando a opção '**Properties**', conforme mostra a Figura 51.

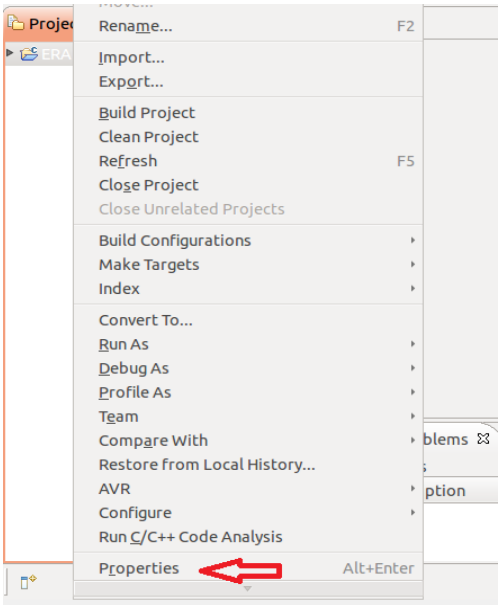

Figura 51 – Acessando as propriedades
Fonte: AUTOR

A primeira configuração a se fazer é configurar o microcontrolador que será utilizado. Para isso escolhe-se a opção **AVR->Target Hardware.** Em seguida escolhe-se o microcontrolador e a frequência utilizada no mesmo. A Figura 52 ilustra esse passo.

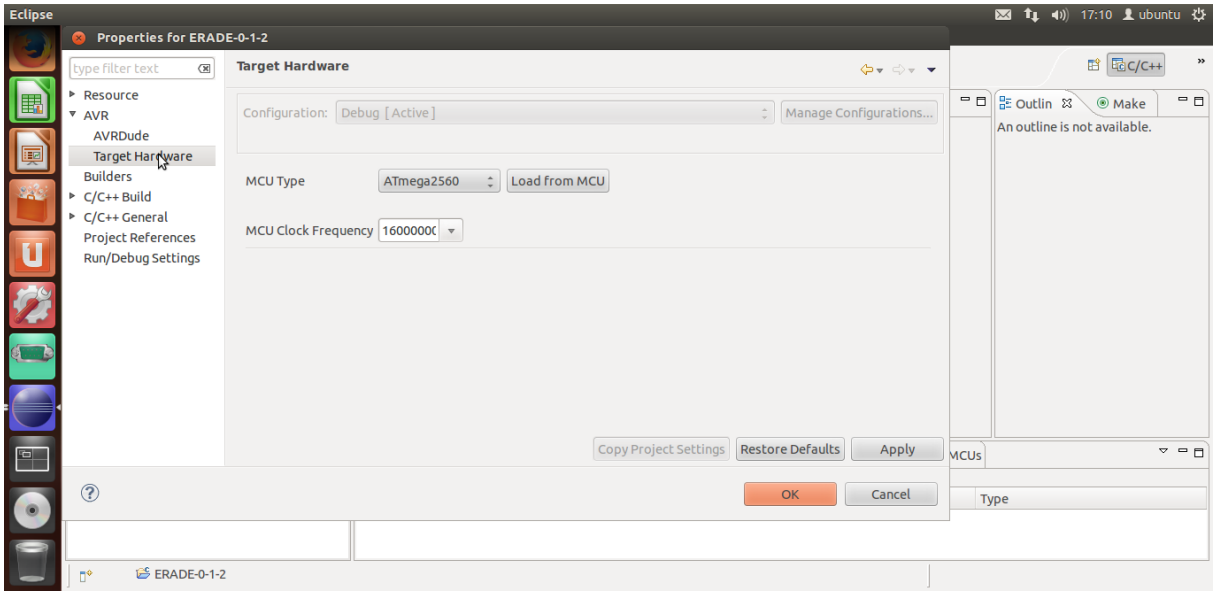

Figura 52 – Configurando o *hardware*
Fonte: AUTOR

Ainda nas configuração, é necessário configurar as opções de 'Include' dos compiladores. Para isso deve-se ir em **C/C++ Build->Settings**. É necessário que a configuração seja realizada tanto para o compilador C quanto para o C++. Por isso se deve selecionar a opção **AVR Compiler->Directories** e adicionar todas as 3 pastas com nome 'include' que existem no projeto do *framework*. A Figura 53 e 54 representam essa etapa. A mesma configuração deve ser feita para a opção **AVR Compiler C++->Directories.** Conforme a Figura 55.

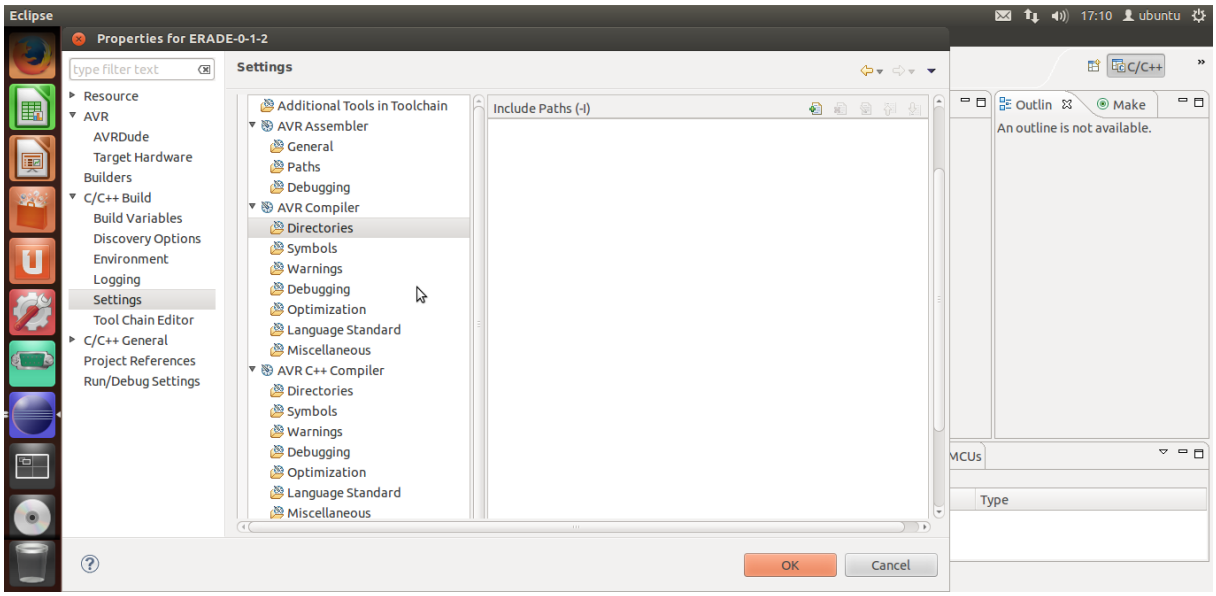

Figura 53 – Configurando includes
Fonte: AUTOR

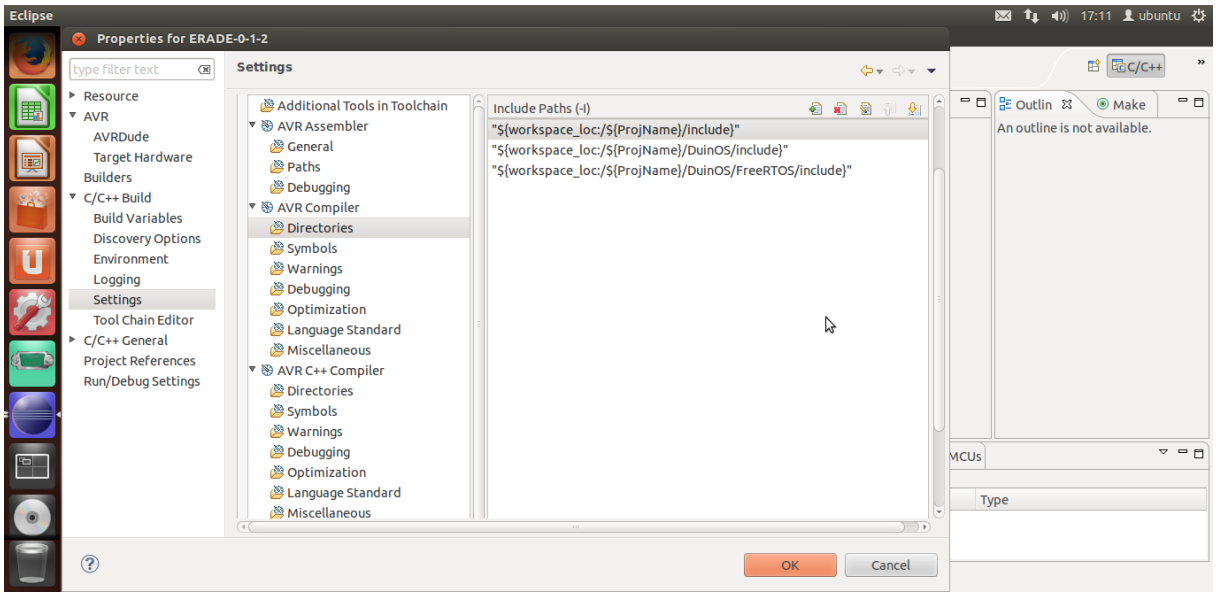

Figura 54 – Configurando diretórios para o compilador C
Fonte: AUTOR

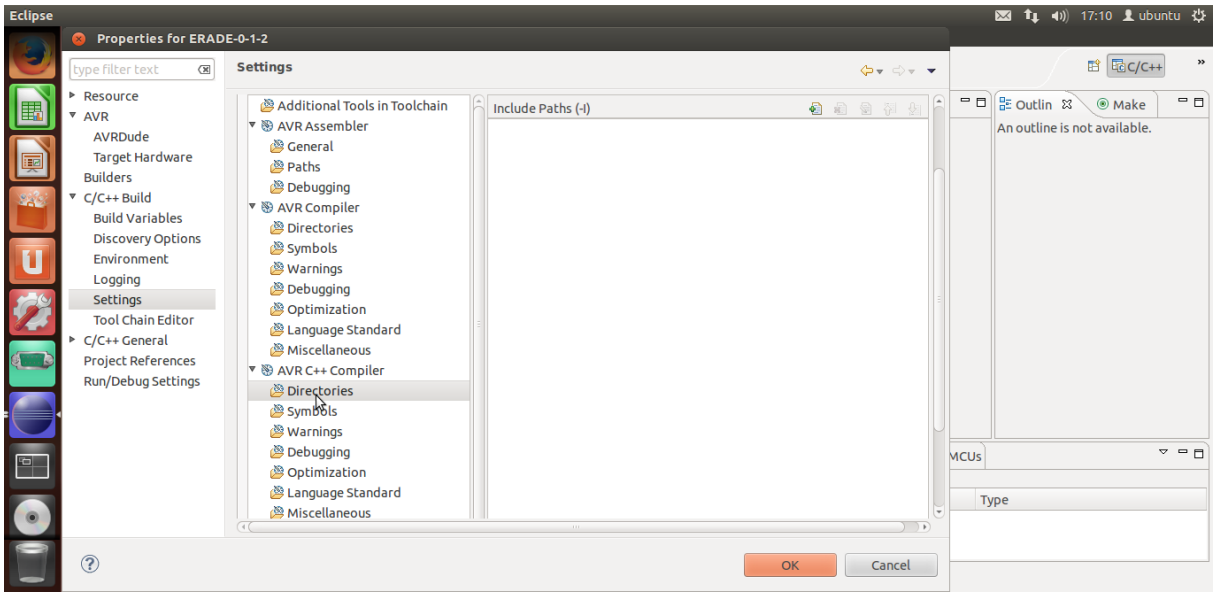

Figura 55 – Configurando diretórios para compilador C++
Fonte: AUTOR

Realizado os passos descritos até aqui, já é possível compilar o projeto. É necessário clicar com o botão direito e escolher a opção **Build Project.** Isso pode levar alguns segundos, enquanto isso ocorre uma janela mostrará o andamento do processo, conforme pode ser visto na Figura 56.

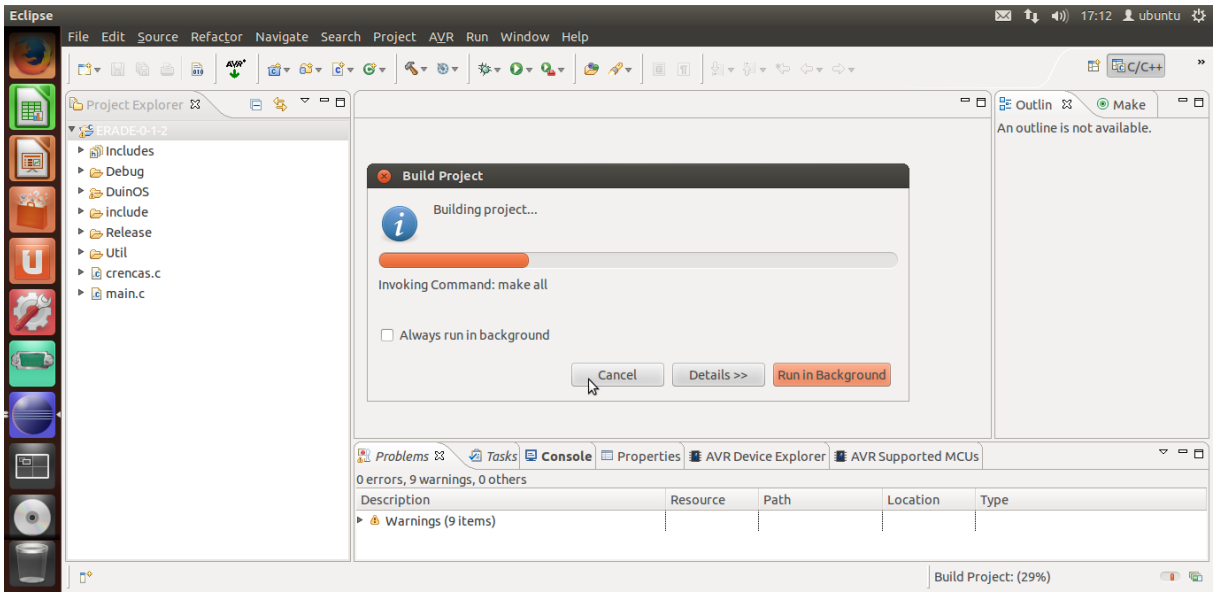

Figura 56 – Compilando o *framework*
Fonte: AUTOR

Após a compilação, conforme mostra a Figura 57, é possível ver na guia **Console** um resumo de uso de memória do microcontrolador calculado pelo compilador. E conforme a

Figura 58, é possível ver que é criada uma pasta DEBUG e RELEASE no projeto. Dentro dessa última vê-se um arquivo com a extensão '.a' e com o nome do projeto começando com o sufixo 'Lib'. Este é o produto do *framework* compilado e é o que é utilizado como biblioteca estática nos projetos. Portanto esse arquivo é o resultado de todo o código do *framework* em um único arquivo.

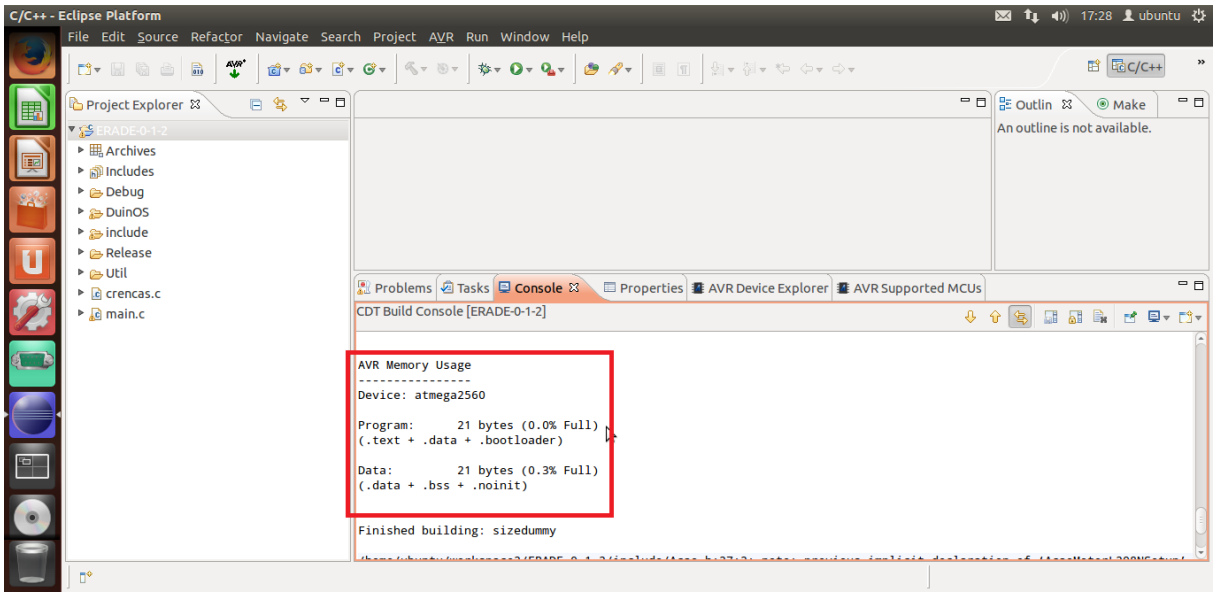

Figura 57 – Estimativas de uso de memória
Fonte: AUTOR

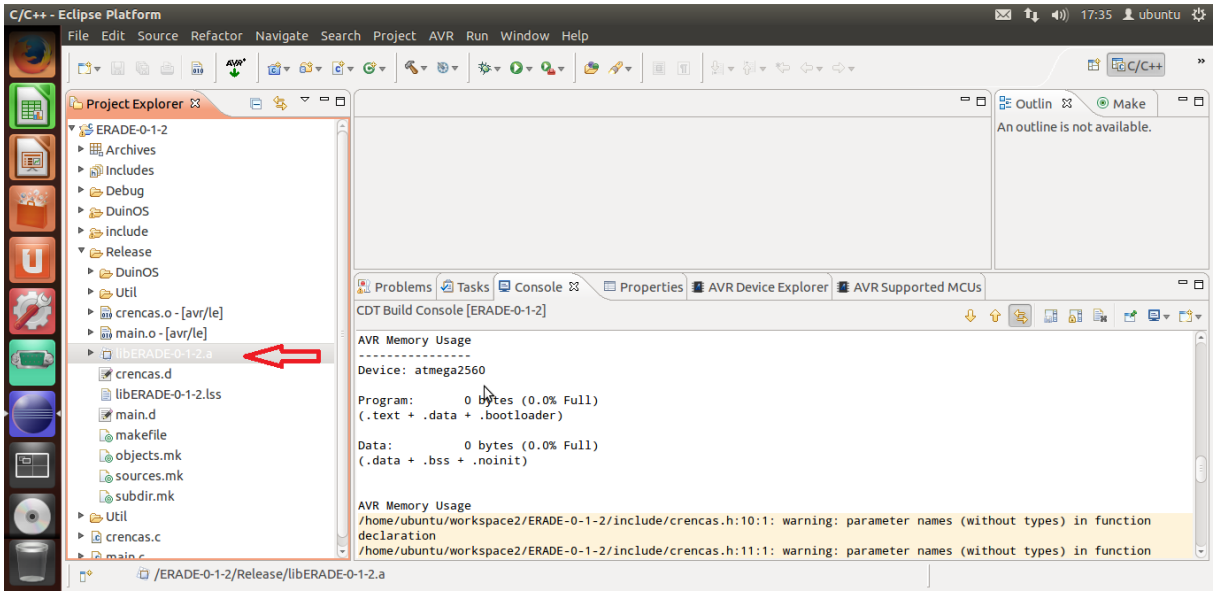

Figura 58 – *Framework* compilado
Fonte: AUTOR

5.2.2 Criando o projeto piloto

Uma vez que já se tem o projeto do *framework* devidamente criado e configurado. Já é possível criar o projeto piloto. A criação deve ser realizada conforme a Figura 59. É importante ressaltar que o tipo de projeto escolhido deve **AVR Cross Target Application**, como mostra a Figura 60.

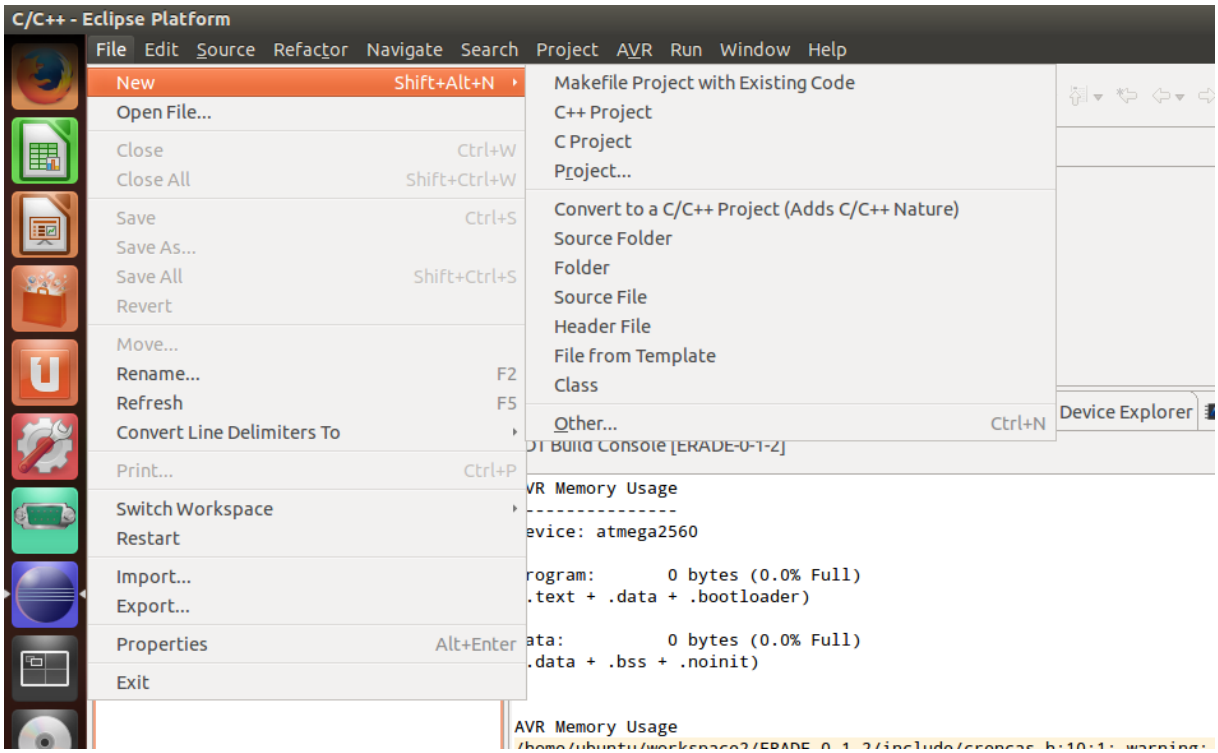

Figura 59 – Começando um novo projeto
Fonte: AUTOR

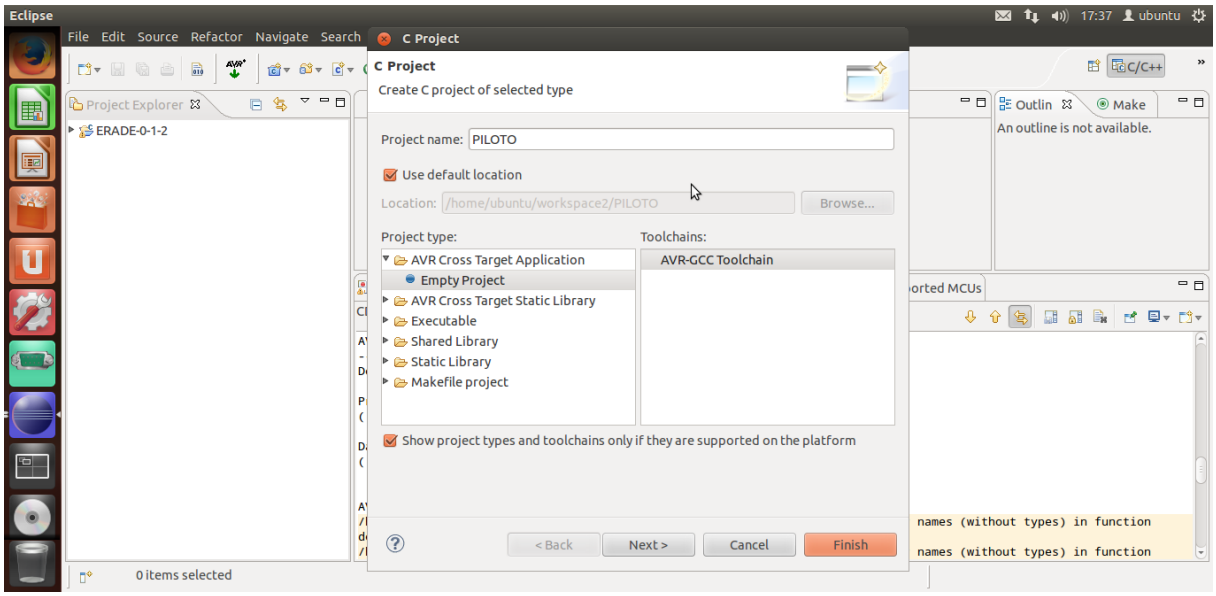

Figura 60 – Definindo nome do projeto piloto
Fonte: AUTOR

Após a criação do projeto deve-se fazer a configuração do microcontrolador utilizado, conforme foi realizado no projeto do *framework*. A Figura 61 ilustra essa configuração.

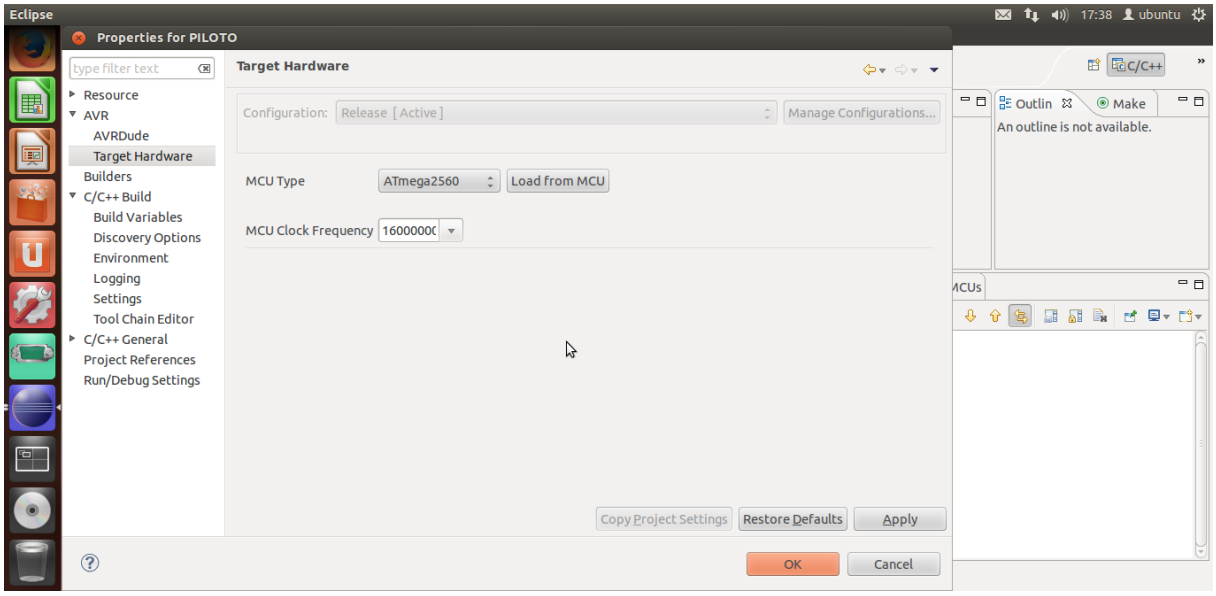

Figura 61 – Configurando *hardware* do projeto piloto
Fonte: AUTOR

Da mesma forma, é necessário realizar a configuração de 'includes' do projeto, apontando para as pastas do projeto do *framework*. Isso pode ser visto na Figura 62.

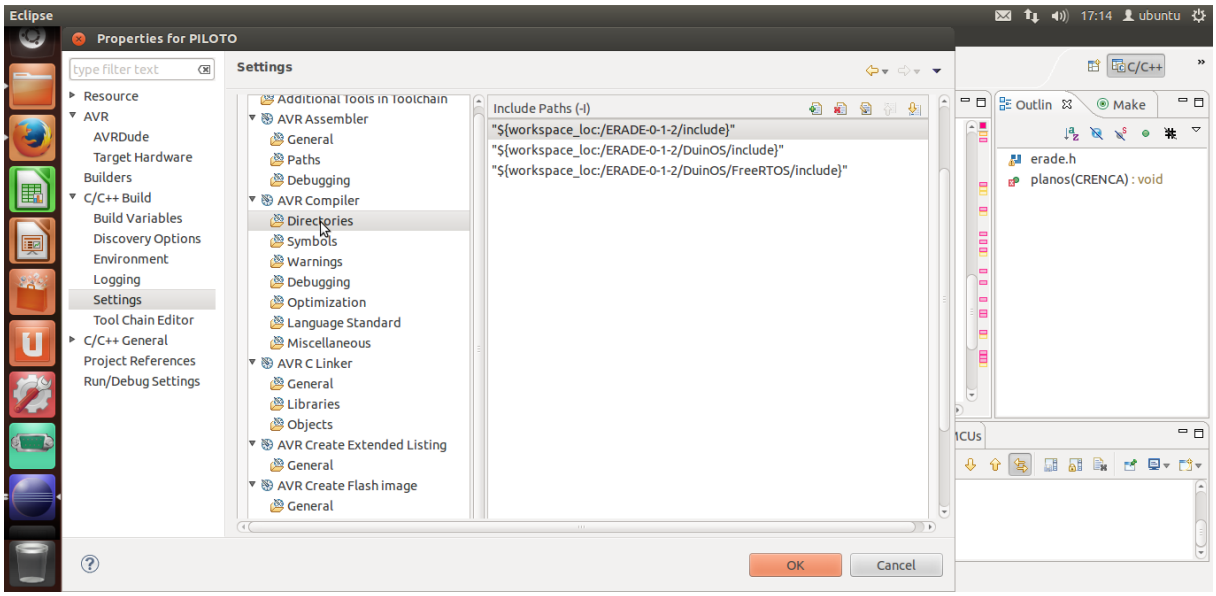

Figura 62 – Configurando diretórios do compilador C para o projeto piloto
Fonte: AUTOR

Neste projeto é necessário configurar o C Linker. Não entra-se em detalhes aqui sobre o funcionamento desse módulo. Para realizar essa configuração deve-se ir em **C/C++**

Build->Settings->AVR C Linker->Libraries e adicionar o caminho de todas as 3 pastas 'Include' do *framework*. Esse passo é visto na Figura 63**.**

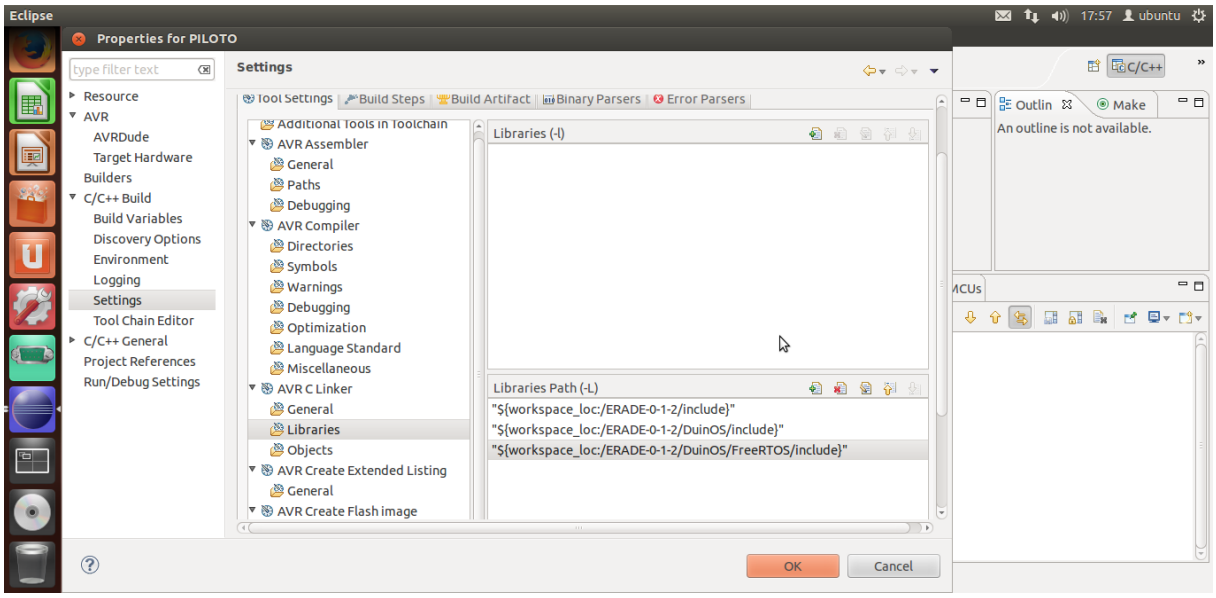

Figura 63 – Configurando pastas do C Linker
Fonte: AUTOR

Após isso é necessário apontar o objeto que foi criado na compilação do *framework*, ou seja o arquivo com a extensão '.a'. Para realizar isso deve-se ir em **C/C++ Build->Settings->AVR C Linker->Objects** e adicionar o caminho onde está o arquivo gerado pela compilação do *framework*. As Figuras 64 e 65 mostram como esse processo deve ser realizado.

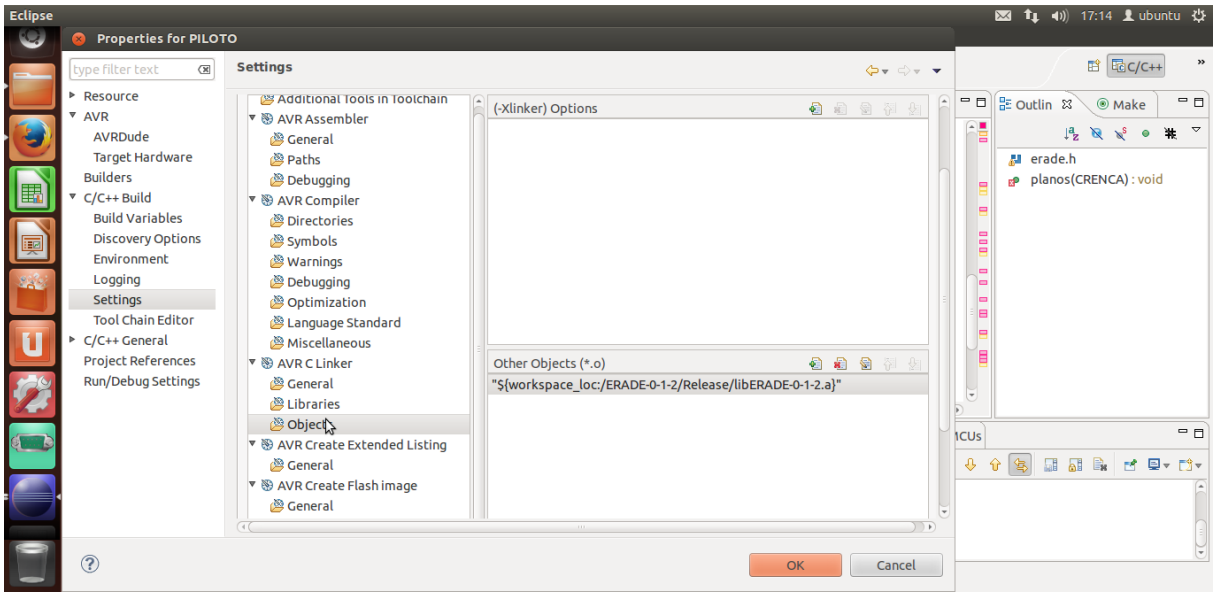

Figura 64 – Configurando objeto no C Linker
Fonte: AUTOR

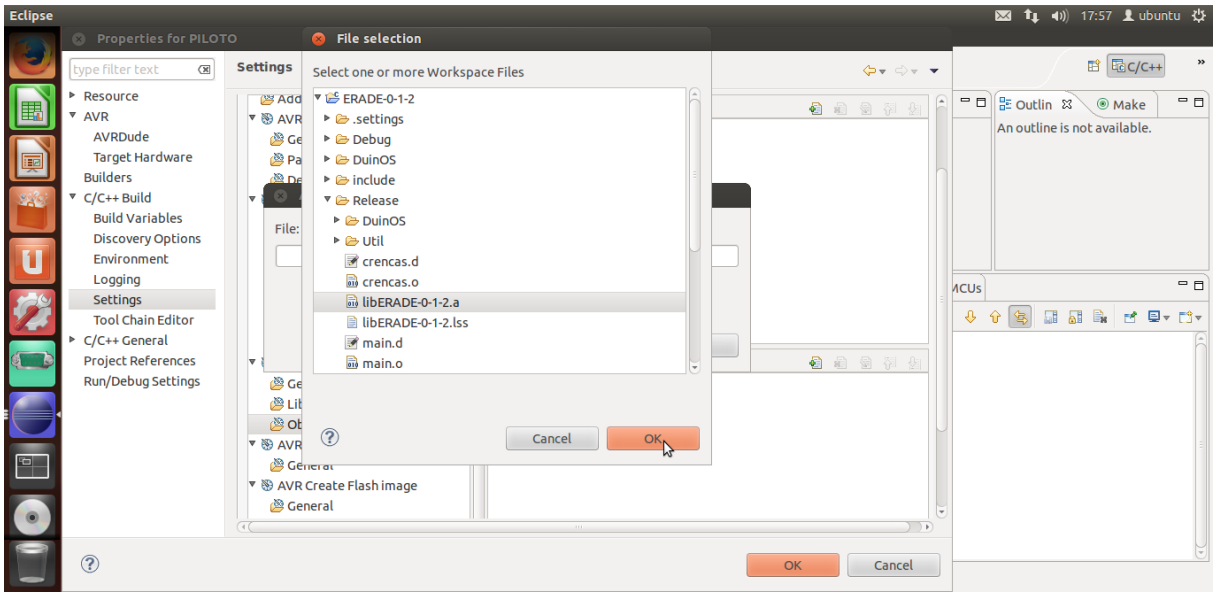

Figura 65 – Escolhendo o arquivo de objeto do C Linker
Fonte: AUTOR

Realizadas as etapas até aqui, já se tem o projeto pronto para uso. No capítulo a seguir explica-se como foi realizada a codificação do projeto piloto.

5.2.3 Codificação

A codificação do projeto deve se iniciar pela definição dos endereços dos sensores, atuadores e das crenças. Essa configuração é realizada no arquivo “EradeConfig.h” que está localizado na pasta ‘include’ na raiz do projeto do *framework*. A Figura 66 mostra a configuração realizada no projeto.

```
#ifndef ERADECONFIG_H
#define ERADECONFIG_H

#include "erade.h"

//sensores habilitados
#define E_FOGO  200
#define E_DISTANCIA 201

//Atuadores habilitados
#define A_MOTOR_L298 1001
#define A_SIRENE 1002

//Comunicação
#define COMUNICACAO_SERIAL 1

//////////////////// CRENCAS /////////////////

//crencas variaveis
#define CA_PERTO (ler(E_DISTANCIA) < 15)
#define CA_FOGO (ler(E_FOGO) != 0)

//crencas do agente
#define TEMFOGO 51

//crenca de subplanos
#define APROXIMAR 52

//endereço da ultima crença ocorrida
#define ULTIMACRENCA 150

//utilizando debug
//#define _DEBUG 1

#endif
```

Figura 66 – Arquivo EradeConfig.h
Fonte: AUTOR

Como pode ser visto na Figura 66, a primeira configuração que é realizada é as dos sensores habilitados. Por convenção todo o endereço de sensor começa com a letra ‘E’ e deve possuir um valor de 200 até 255. Este valor se deve ao fato de otimizar o espaço de armazenamento, pois é necessário reservar apenas 1 Byte para armazenar a crença. Outro detalhe importante é que os sensores estão diretamente ligados a biblioteca construídas para eles, ou seja, mesmo que aqui exista a configuração do endereço do sensor de ambiente é necessário que a biblioteca também utilize essa definição. A mesma configuração ocorre para os atuadores, porém aqui a convenção de endereçamento é sempre utilizar números maiores que 1000. Isso deve-se ao fato de se imaginar que não seja necessário armazenar uma crença

sobre um atuador. Outra convenção utilizada foi o sufixo ‘A’ na frente dos nomes dos atuadores. A configuração da comunicação é realizada apenas com valores *boolean*, assim está habilitado ou não. Na versão atual existe apenas a comunicação serial disponível para utilização.

As crenças do agente como dito anteriormente são endereços que armazenam valores verdadeiros e falsos, porém existem casos em que a crença é relativa a uma ou mais variáveis. Um exemplo disso é imaginar uma crença que defina se o agente está longe ou perto de algo. Pode se definir um plano para controlar isso, mas isso pode ser considerado um tanto desnecessário, por isso se optou pela geração de crenças variáveis ou relativas, que diferente das crenças convencionais não precisam ser armazenadas, pois apenas refletem expressões lógicas. Um exemplo disso é a crença criada CA_PERTO, que nada mais é que a verificação se a distância é superior a 15 cm. O único inconveniente de utilizar crenças variáveis é o fato de não se poder utilizar a crença como um evento disparador em um plano. Para fazer isso deve-se criar uma crença convencional e criar um plano para fazer essa verificação. A convenção utilizada nas crenças variáveis é a utilização do sufixo ‘CA’. As demais crenças podem ser criadas utilizando valores de 1 a 199, apenas não podendo ter o número 150, que foi reservado para guardar a última crença que sofreu alteração. A última crença é a crença que é utilizada como evento disparador do planos.

```
#include <erade.h>

void planos(CRENCA crenca){

    switch (crenca) {
            case 0:
                // não tem nada, procura
                if(!consultaCrenca(TEMFOGO) && !CA_PERTO ){
                    Direita(300);
                    Frente(1000);
                }else

                //cuidado para nao bater em nada!!
                if(!consultaCrenca(TEMFOGO) && CA_PERTO ){
                    Atras(7000);
                    Direita(300);
                }
                break;

            case E_FOGO:
                //achou fogo e esta longe,acredita que tem fogo na direcao
                if ((!CA_PERTO) && (CA_FOGO ))
                {
                    incluirCrenca(TEMFOGO);
                    A_notificar();
                }else

                //perdeu o fogo de vista, deve reinciar a busca
                if(!CA_FOGO )
                {
                    Esquerda(1000);
                    retirarCrenca(TEMFOGO);
                }
                break;

            case TEMFOGO:

                //tem fogo na direcao, avanca até achar
                if(CA_FOGO && !CA_PERTO){
                    SubPlanos(APROXIMAR);
                }else

                //ACHOU O FOGO
                if(CA_FOGO && CA_PERTO){
                    TocarSirene(1000);
                }else

                //NÃO ACHOU MAIS O FOGO
                if(!CA_FOGO){
                    retirarCrenca(TEMFOGO);
                    gravar(ULTIMACRENCA, 0);
                }

            default:
                break;
        }
}
```

P1 P2 P3 P4 P5 P6 P7

Figura 67 – Planos do projeto piloto
Fonte: AUTOR

Qualquer projeto que venha a utilizar o *framework* proposto deve possuir no mínimo 2 arquivos. Um arquivo chamado ‘Planos.c’ e o outro ‘SubPlano.c’. O arquivo Planos.c do projeto piloto é mostrado na Figura 67. É possível ver que ele segue a estrutura de planos proposto no capítulo 5.2. Também é possível ver que existem 7 planos escritos. Abaixo é explicada a lógica de cada um deles.

- P1: é possível ver que esse plano possui como evento disparador o valor 0, isso quer dizer que ele pode ser executado quando o sistema for iniciado, desde que o contexto do plano seja considerado verdadeiro. O contexto desse plano possui duas sentenças, uma é a consulta de que a crença TEMFOGO é falsa. A outra verifica a crença variável CA_PERTO para verificar se está ou não perto de algo. Caso o contexto seja verdadeiro são executado duas tarefas, uma que é andar para a direita e a outra é andar para frente. Ambas funções fazem parte da biblioteca de ação ‘Acao-MotorL298N.h’. Nota-se que os valores passados nas duas funções são diferentes, isso porque a intenção aqui é que o robô ande fazendo trajetos ovais.
- P2: o segundo plano também pode ser executado já na inicialização do sistema, mas o contexto desse plano é a verificação se o agente está perto de algo e se ainda não achou fogo. Esse plano tem como objetivo fazer com que o robô não colida com objetos e paredes, por isso as ações do plano são para andar para atrás e para a direita, desviando assim do possível obstáculo.
- P3: este plano tem com disparador a crença E_FOGO. Essa crença muda quando o fogo é encontrado ou perdido de visão. Este plano possui duas condições para execução. A primeira verifica se o fogo está longe e a segunda se o fogo ainda está visível. Caso o plano seja executado ele incluirá a crença que existe fogo e emitirá uma notificação pela serial do microcontrolador. Certamente essa notificação apenas é enviada se existir comunicação serial habilitada.
- P4: este plano também é disparado pelo evento E_FOGO, porém aqui o contexto verifica se o fogo não está mais visível, caso não esteja o plano é executado. As ações aqui são andar para a esquerda, isso na esperança de que apenas o robô passou pelo fogo quando fazia sua trajetória oval. A outra ação é retirar a crença de que existe fogo.
- P5: o quinto plano possui como evento disparador a crença TEMFOGO, tanto quando é tornada verdadeira, quanto ela é assinalada com falsa, portanto os plano 3 e 4, são

antecessores desse plano. O contexto do plano verifica se ainda existe fogo e se está longe, caso isso seja verdadeiro o plano executa um sub-plano chamado APROXIMAR. Os sub-planos são vistos a seguir.

- P6: este plano possui o mesmo evento disparador do P5. O contexto aqui verifica se existe fogo e se está próximo, assim pode-se dizer que o plano em questão é o plano de sucesso, pois o fogo foi localizado. Sendo assim a ação aqui é disparar uma sirene de alerta.
- P7: o último plano tem como disparador a crença TEMFOGO, ou seja em algum momento o fogo foi notado, porém o contexto do plano verifica se ainda existe fogo, caso realmente não exista, a crença do fogo é retirada e a busca é reiniciada.

```
#include <erade.h>

void SubPlanos(CRENCA crenca){

    switch (crenca) {
        case APROXIMAR:

            //TEM FOGO NA DIREÇÃO, AVANÇA
            if(CA_FOGO && !CA_PERTO){
                Frente(300);
                SubPlanos(APROXIMAR);
            }
            break;


        default:
            break;
    }
}
```

Figura 68 – Sub-planos do projeto piloto
Fonte: AUTOR

O arquivo ‘SubPlanos.c’ pode ser visto na Figura 68. Neste projeto apenas se utilizou o sub-plano APROXIMAR. Este sub-plano nada mais faz de que realizar uma aproximação do fogo. Tem ele como contexto a verificação se o fogo está visível e se está longe. Caso as duas sentenças seja verdadeiras a ação executada é seguir em frente. De forma recursiva o

sub-plano é realizado até que o contexto não seja mais verdade, ou seja, o agente já está perto do fogo ou o fogo já não é mais pressentido.

5.2.4 Funcionamento

Após realizar a codificação dos planos, já possível realizar a compilação do projeto. Lembrando que sempre se deve compilar o *framework* e em seguida o projeto. Após a compilação é mostrado no console da IDE uma estimativa de consumo de memória, obviamente somente se houve sucesso na compilação. A Figura 69 mostra isso.

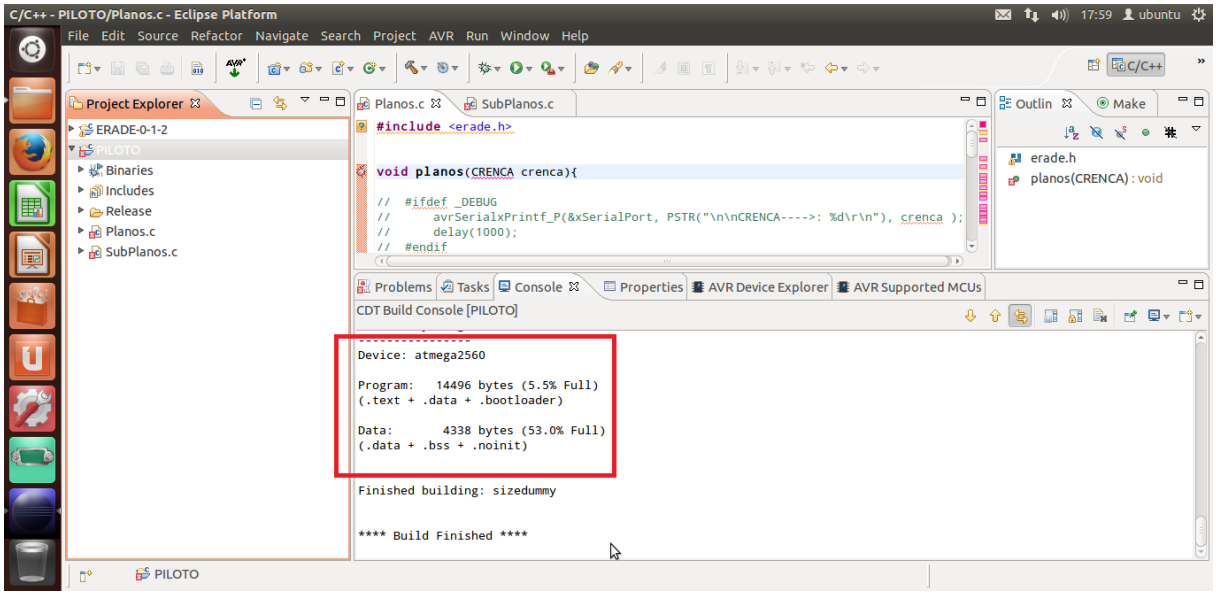

Figura 69 – Resultado da compilação do projeto piloto
Fonte: AUTOR

Neste momento já é possível fazer a gravação do código no *hardware*. Neste projeto piloto se utilizou o microcontrolador ATmega2650 com a placa Arduino Mega. Para realizar a gravação do microcontrolador utilizou-se a ferramenta AVR-Dude, porém conforme já mencionado não entra-se aqui em detalhes técnicos sobre o *plug-in*, pois além dessa ferramenta de gravação de micocontroladores existem muitas outras no mercado. Contudo, os passos de configuração do projeto serão apresentados aqui.

Nas propriedades do projeto deve-se ir em **AVR->AVRDude**. Como não existe nenhuma configuração de programação é necessário criar uma através do botão **New**, como pode ser visto na Figura 70**.**

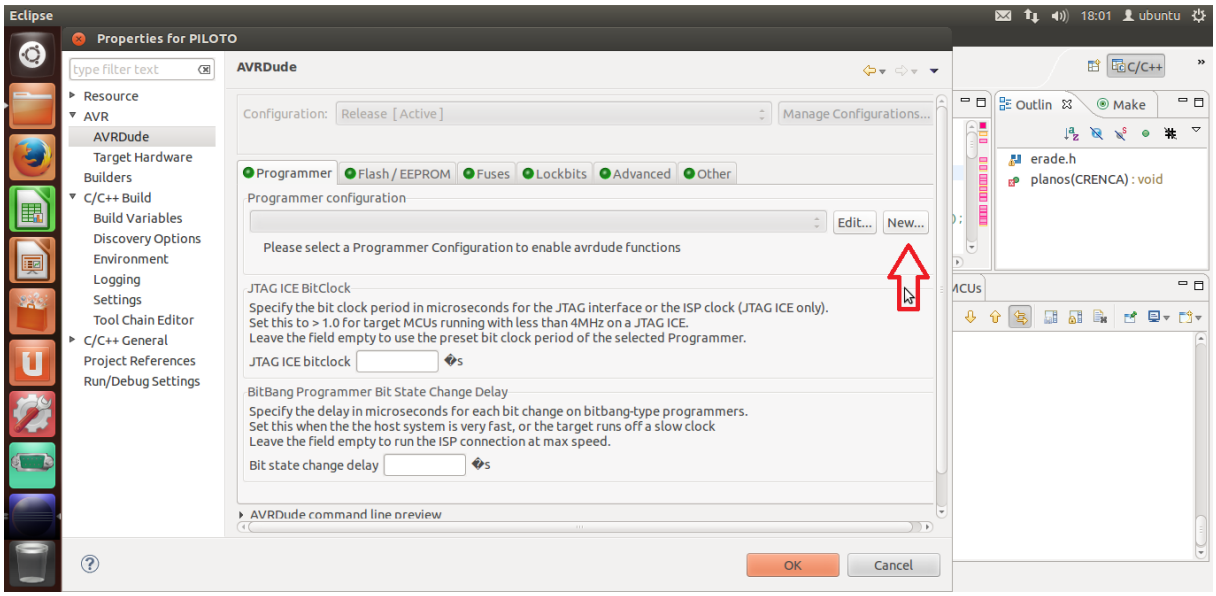

Figura 70 – Criando uma configuração de gravador
Fonte: AUTOR

Existem 3 configurações essenciais no gravador, a primeira é o *hardware* de comunicação, a segunda é a porta de comunicação e a terceira é a velocidade na comunicação. A Figura 71 mostra essas 3 configurações respectivamente. Realizadas essas configurações, é possível fazer a gravação.

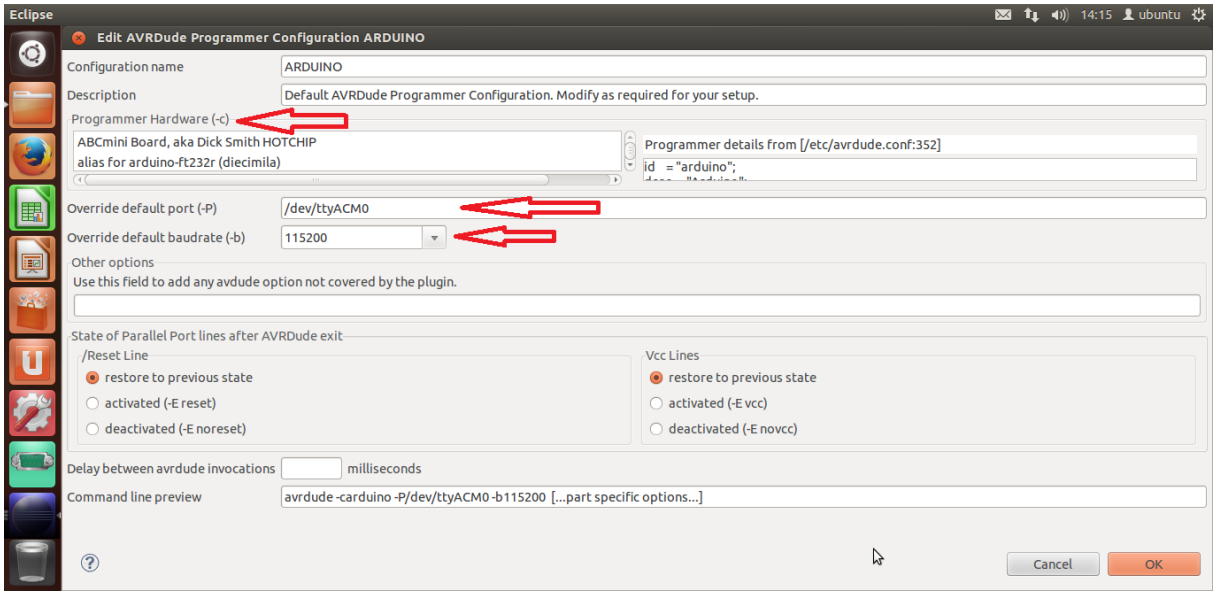

Figura 71 – Configurando um gravador
Fonte: AUTOR

Para gravar o microcontrolador apenas é necessário conectar o mesmo no computador através da interface escolhida e utilizar o botão ‘AVR’, na barra de ferramentas, conforme mostra a Figura 72.

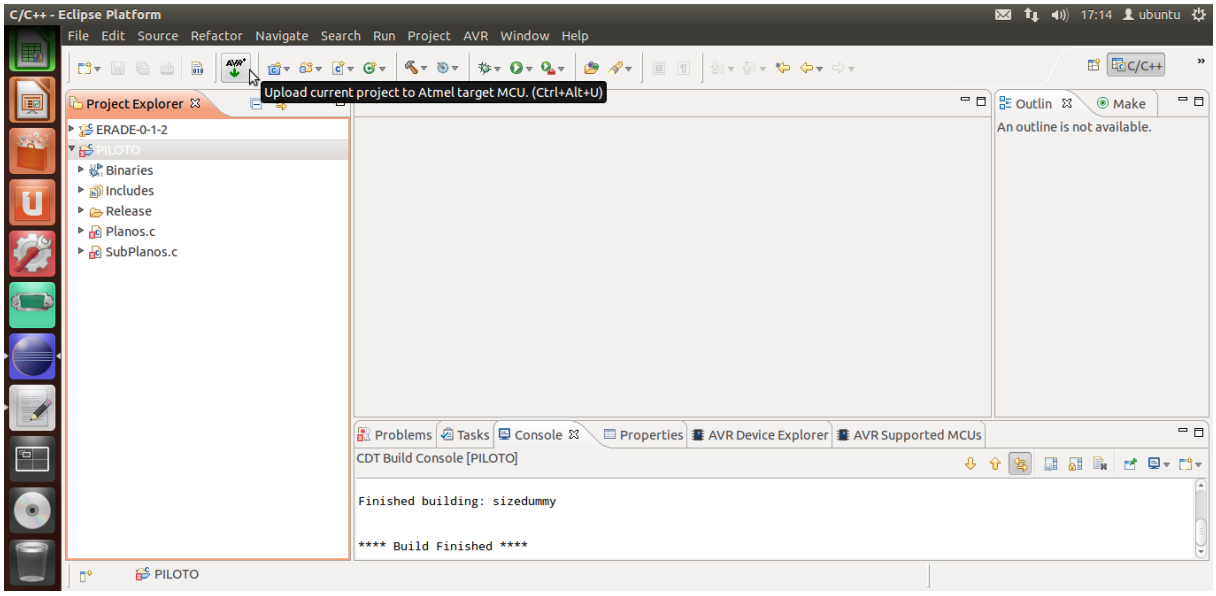

Figura 72 – Gravando o microcontrolador
Fonte: AUTOR

Realizado esse passo, já se tem o robô pronto com o *hardware* montado e agora com o *software* embarcado no microcontrolador. Então já é possível realizar o teste prático com ele. A Figura 73 mostra uma sequência de imagens do robô em execução.

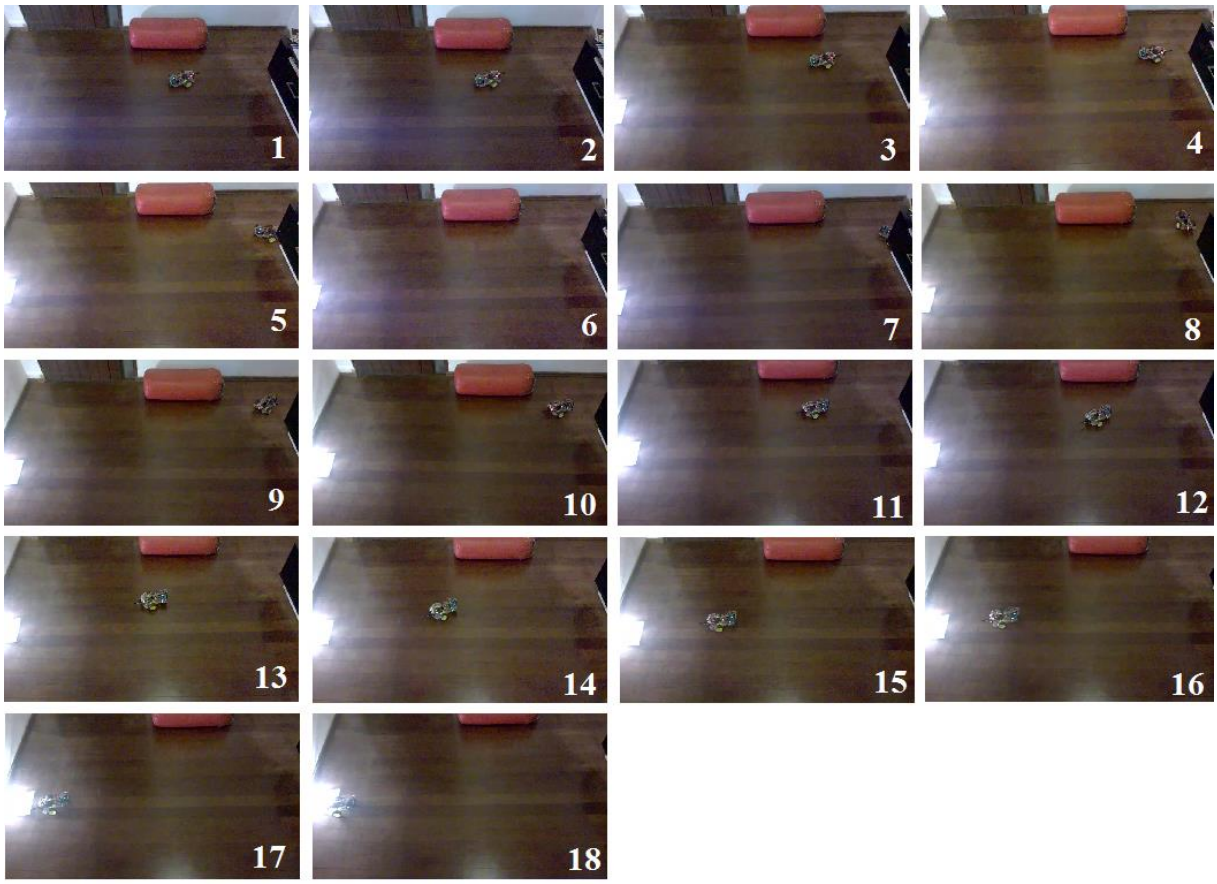

Figura 73 – Robô em funcionamento
Fonte: AUTOR

O teste mostrado na Figura 74 foi realizado em um ambiente de aproximadamente 2,5 x 2,5 metros. Para simulação do fogo foi utilizado uma lâmpada encandeceste de 60W. Ela foi colocada na parte esquerda do ambiente. Já o robô foi colocado propositalmente virado para a direita e em direção a um canto onde existem 2 obstáculos, para que assim seja possível testar se o robô realmente não ficará trancado em cantos ou obstáculos. Nas imagens de 1 a 4, é possível ver o robô indo em direção ao canto mencionado. Nas imagens 5 a 8, é possível ver robô entrando no campo e conseguindo sair facilmente. Ao sair do canto, o robô anda para a esquerda e sente a chama, neste momento ele começa a aproximação que ocorre em linha reta, como pode ser visto nas imagens de 10 a 17. Ao chegar na frente da lâmpada a sirene é disparada, portanto o robô obtém o sucesso.

O detalhamento da execução do robô pode ser visto no Quadro 3. Neste Quadro é possível ver na coluna imagem, a etapa correspondente a Figura 73. A coluna ‘Plano em execução’ faz referência aos planos da imagem 67. As demais colunas mostram o estado das crenças do agente, tanto as crenças ditas como comuns, como as variáveis.

Quadro 4 – Robô em funcionamento
Fonte: AUTOR

Imagem	Plano em execução	Crenças	Crenças Variáveis	
		TEMFOGO	CA_FOGO	CA_PERTO
1	P1	FALSA	FALSA	FALSA
2	P1	FALSA	FALSA	FALSA
3	P1	FALSA	FALSA	FALSA
4	P1	FALSA	FALSA	FALSA
5	P1	FALSA	FALSA	FALSA
6	P2	FALSA	FALSA	VERDADEIRA
7	P2	FALSA	FALSA	VERDADEIRA
8	P1	FALSA	FALSA	FALSA
9	P3	VERDADEIRA	VERDADEIRA	FALSA
10	P5	VERDADEIRA	VERDADEIRA	FALSA
11	P5	VERDADEIRA	VERDADEIRA	FALSA
12	P5	VERDADEIRA	VERDADEIRA	FALSA
13	P5	VERDADEIRA	VERDADEIRA	FALSA
14	P5	VERDADEIRA	VERDADEIRA	FALSA
15	P5	VERDADEIRA	VERDADEIRA	FALSA
16	P5	VERDADEIRA	VERDADEIRA	FALSA
17	P5	VERDADEIRA	VERDADEIRA	FALSA
18	P6	VERDADEIRA	VERDADEIRA	VERDADEIRA

5.3 Validação e análise

A proposta do trabalho nunca foi construir um robô detector de chama, porém quando o robô se mostra funcional, é possível ver o sucesso do real objetivo do trabalho, que foi a elaboração de uma proposta de desenvolvimento de sistemas embarcados com características de um agente inteligente, mais especificamente de um agente reativo. Contudo mesmo com o sucesso no desenvolvimento da proposta é necessário realizar uma análise mais detalhada sobre os prós e os contras da utilização da proposta. O Quadro 4 mostra uma comparação de um projeto utilizando o *framework* proposto e um desenvolvimento convencional.

Quadro 5 – Análise de desenvolvimento
Fonte: AUTOR

	Desenvolvimento convencional	Com framework
Uso de memória de armazenamento	MENOR	MAIOR
Uso de memória de execução	MENOR	MAIOR
Uso de processamento	IGUAL	IGUAL
Decomposição hierárquica	NÃO	SIM
Tamanho código fonte	MAIOR	MENOR
Reutilização de código	MENOR	MAIOR

O uso de memória é claramente mais alto utilizando o *framework* proposto, isso vale tanto para a memória estática de armazenamento quando para a memória de execução. A memória estática é maior devido a utilização de várias bibliotecas separadas, tanto para controlar ações como para controlar os sensores, isso sem mencionar o próprio RTOS que foi utilizado.

Já para a memória de execução o uso é maior principalmente devido ao RTOS, pois quando há uma troca de contexto entre tarefas, o estado atual da tarefa é armazenado em memória, assim quando o escalonador retorna a esta tarefa o seu estado é iniciado de onde parou. Apesar de o *framework* utilizar apenas quatro tarefas e o tamanho que cada tarefa pode alocar possa ser configurável, ainda assim é evidente que exista um uso maior de memória.

A decomposição hierárquica se mostra muito maior no projeto utilizando o *framework*. Isola-se as camadas mais baixas da aplicação fazendo com que na codificação dos planos em nenhum momento seja necessário realizar a chamada de funções de acesso a *hardware* para verificação de estado de sensores. O exemplo utilizado para testar foi simples utilizando apenas 2 sensores, mas poderia ter 10 sensores por exemplo, que a maneira com que os planos foram escritas não mudaria, apenas se teria mais crenças de ambiente para utilizar, deixando os planos ainda mais precisos.

Se considerado apenas o código utilizado para o projeto específico é visto uma redução em relação ao desenvolvimento convencional, isso porque somente é necessário codificar os planos e sub-planos para cada projeto, pois todo o resto do código faz parte do *framework*, mostrando assim um reaproveitamento de código imenso. As bibliotecas de ambiente e ações podem ser comuns aos projetos, inclusive podem ser retiradas bibliotecas não utilizadas para diminuir o tamanho do código compilado.

A comparação feita aqui é realizada em relação ao desenvolvimento de uma sistema embarcado apenas utilizando o conceito de *super-loop*. Para sistemas com RTOS a comparação pode ser muito relativa, pois pode variar muito devido ao sistema utilizado, mas é evidente que apenas utilizar um RTOS não proporcionaria as vantagens geradas pelo *framework*. Obviamente também o *framework* no estágio atual ainda é prematuro, principalmente em relação as bibliotecas disponíveis, porém apresenta-se promissor o resultado obtido com a sua utilização no projeto piloto.

O projeto foi testado em microcontroladores de 2Kbytes e 8Kbytes de SRAM. Como esperado, em microcontroladores de memória de 2Kbytes, cerca de 70% da memória é comprometida com o *framework*. Isso gera um inconveniente, pois limita principalmente alguns tipos de sensores utilizados, pois algumas bibliotecas necessitam de mais memória de execução.

Durante os testes também foram percebidos problemas no uso de alguns tipos de armazenamento, como por exemplo em cartões SD. Isso ocorre porque os cartões geralmente são formatados em formato FAT, por isso possuem blocos de 512 bytes, ou seja um buffer mínimo para se utilizar esse recurso é de 512 bytes. Portanto se o microcontrolador possui 2 Kbytes e o uso do *framework* aproxima-se de 70%, sobram apenas um pouco mais de 400 bytes. Sendo assim impossível utilizar armazenamento SD para as crenças em microcontroladores de 2Kbytes utilizando o *framework*. Em microcontroladores de 8Kbytes não foram constados nenhum tipo de problema com utilização de memória.

6 CONCLUSÃO

Cada vez mais é possível ver trabalhos utilizando a modelagem de agentes para ambientes complexos. Combinando sistemas embarcados com a modelagem de agentes é possível desenvolver equipamentos capazes de interagir nesses ambientes e realizar tarefas úteis à vida humana moderna. O presente trabalho apresentou uma proposta para que essa combinação seja possível, sendo esse sua principal contribuição para essa área de estudo. Obviamente, como visto no decorrer do trabalho, os 2 temas são bem distintos em si, gerando assim diversos problemas numa tentativa de junção harmoniosa. Alguns foram solucionados durante o trabalho e outros ficam para trabalhos futuros.

No decorrer do trabalho foram realizadas diversas pesquisas bibliográficas no intuito de conduzir a implementação do *framework* como base em conceitos já consolidados. No entanto diversos momentos houve conflitos com as limitações de *hardware*. Certamente uma solução fácil para isso seria realizar testes com hardwares mais robustos, porém isso conflitaria com algumas premissas dos sistemas embarcados defendidas por Peter Marwedel (2006) e explicadas no capítulo 1. Sendo assim, sempre se optou em otimizar o máximo o uso do *hardware* mesmo que isso sacrificasse em partes a arquitetura do *software*. São em situações assim que é possível compreender a diferença no desenvolvimento de softwares embarcados com softwares convencionais. Isso também explica a complexidade dos sistemas embarcados, uma vez que a arquitetura do *software* se limita até o ponto onde o *hardware* permite. Assim sendo, fica evidente que sua complexidade em parte se gera por uma arquitetura básica e simples que impede que abstrações mais complexas sejam realizadas. No *framework* esse problema foi diretamente percebida na implementação de um padrão para a biblioteca de planos, apresentada no capítulo 4.3. Inicialmente o desenvolvimento começou tentando utilizar um nível de abstração muito elevado, indo ao ponto de que os planos poderiam ser codificados em uma pseudo linguagem, mais próxima de uma linguagem natural. Logo nos primeiros experimentos essa abordagem já se mostrou inválida, pois o consumo de memória na conversão dos planos em uma linguagem que possa ser processada em tempo de execução era muito elevada. Outro problema foi a latência no acesso aos dados, uma vez que a biblioteca de planos não estaria compilada junto do projeto. Após essas conclusões os níveis de abstração foram baixando chegando até ao ponto que se obteve certeza que a melhor solução é a utilização de uma biblioteca de planos estática e compilada junto com o projeto. Problemas dessa natureza apareceram ao longo de todo o trabalho. Em

cada decisão da estratégia de funcionamento do *framework*, foi necessário a comparação dos impactos que seriam gerados na sua adoção. Tantos impactos positivos, geralmente ligados à abstrações de *software*, como nos aspectos negativos, geralmente ligados ao comprometimento excessivo do *hardware*.

Outro grande problema apresentado no desenvolvimento do *framework* foi a questão da depuração do *software*. Como em um sistema embarcado o *software* é diretamente vinculado ao *hardware*, quando é necessário depurar o código fonte é necessário também realizar a depuração junto do *hardware*. Isso se deve ao fato que o *software* age com base nas informações geradas pelo *hardware*. Assim não é possível analisar um comportamento apenas analisando os pontos do código fonte, pois o problema pode estar em sinais eletrônicos no *hardware*. Existem algumas ferramentas que auxiliam neste processo. Essas ferramentas geralmente são equipamentos colocados entre o *hardware* e o computador do desenvolvedor, tornando assim possível verificar, por exemplo, valores de variáveis do *software* e sinais eletrônicos do *hardware* ao mesmo tempo. Porém no desenvolvimento do framework não se tinha uma ferramenta dessa disponível, o que levou a utilização de métodos de depuração mais demorados e menos precisos. Como por exemplo, o monitoramento serial que foi desenvolvido, colocando *breakpoints* manuais em certos pontos do código juntamente com medições realizadas com auxílio de equipamentos de medição eletrônica.

Apesar de todas as dificuldades enfrentadas no desenvolvimento do *framework* proposto o resultado obtido foi gratificante e satisfatório. Mostrando que é possível o desenvolvimento de *hardware* com o conceito de agentes. Todo o código fonte gerado durante o trabalho foi disponibilizado para utilização livre para que mais pessoas possam realizar experimentos e assim contribuírem para o amadurecimento do *framework* e dos conceitos aqui estudados.

6.1 Trabalhos futuros

De forma alguma é possível dizer que o *framework* pode ser considerado como finalizado, porém no estágio atual já se encontra funcional como mostrado no projeto piloto. No entanto, como já dito, este trabalho não se aprofundou na exploração da comunicação entre agentes. Esse aprofundamento não ocorreu por se entender que o tema necessita de um trabalho especifico, somente com esse objetivo. Como visto no capítulo 2.2, a comunicação é extremamente complexa e importante para a socialização de agentes em uma comunidade,

ressaltando também a padronização FIPA que deve ser observado neste processo de comunicação. Sendo assim um dos trabalhos que ficam para realizar no futuro é a implementação de padrões de comunicação entre agentes no *framework*.

Em todo o desenvolvimento do trabalho sempre foi focado a abstração entre as camadas do *framework,* com isso, um outro passo importante que deve ser realizado é o desenvolvimento de uma IDE. Essa IDE deve ser capaz de realizar a codificação dos planos e dos endereços das crenças de um forma visual e amigável. Com isso será possível alcançar um outro nível de desenvolvimento, pois não será necessário conhecimento em uma linguagem de programação para que um indivíduo consiga escrever planos, apenas deve-se conhecer a IDE e de forma visual construa a biblioteca de planos e as demais configurações necessárias para cada projeto.

REFERÊNCIAS BIBLIOGRÁFICAS

ANDRADE, F. S., OLIVEIRA, A. S., '**Sistemas Embarcados – Hardware e Firmware na prática.**', Editora Érica, São Paulo, SP, 2006.

ATZORI L., LERA A., MORABITO, G., '**The Internet of Things: A survey**', Computer Networks, V. 54, Elsevier, (SL), 2010.

AHO, A. V; LAM, M. S; SETHI R.; ULLMAN, J. D; '**Compiladores - Princípios, Técnicas e Ferramentas**'; 2ª Edição; São Paulo; Person Addison-Wesley, 2008.

BARY, Richard; '**Using the FreeRTOS Real Time Kernel - A Practical Guide opened**'; SL; 2009.

BELLIFEMINE, F., CAIRE, G. AND GREENWOOD, D. '**Developing Multi-Agent Systems with JADE**', John Wiley & Sons Ltd, England. 2007.

BRATMAN, M. '**Intention, Plans, and Practical Reason**'. Harvard University Press. 1987.

BORDINI, R.H; HÜBNER, J. F; VIEIRA, R. '**Introdução ao desenvolvimento de sistemas multiagentes com Jason**'. S.L., 2004. Disponível em:.< http://jomi.das.ufsc.br//pubs/2004/Hubner-eriPR2004.pdf> Acesso em: 07/05/2016.

BORDINI R. H; HÜBNER J. F; WOOLDRIDGE, M; '**Programming Multi-Agent Systems In Agentspeak Using Jason**'; Wiley-Interscience; University of Liverpool, UK; 2007

CARRO, L; WAGNER, F., '**Sistemas Computacionais Embarcados**'. In: Jornadas de Atualização em Informática. Campinas, Brasil: Sociedade Brasileira de computação, 2003, P.45-94.

DATASHEET 382p, ATMEL, Disponível em: <http://www.atmel.com/images/atmel-8271-8-bit-avr-microcontroller-atmega48a-48pa-88a-88pa-168a-168pa-328-328p_datasheet_complete.pdf >, Acessado em 23 de novembro de 2015.

ERADE, GitHub Inc, Disponível em: < https://github.com/EvertonJSilva/ERADE>, Acessado em 31 de maio de 2016.

GEORGEFF, M. P.; RAO, S. A; SINGH, P. M., '**Formal Methods in DAI: Logic-Based Representation and Reasoning**' In: Multiagent Systems - A Modern Approach to Distributed Modern Approach to Artificial Intelligence, P 331-376, The MIT Press, Cambridge, Massachusetts; London, England, 1999.

GUERRA-HERNÁNDEZ, A; ORTIZ-HERNÁNDEZ, G., '**Towards BDI sapient agents: learning intentionally**' in: Toward Artificial Sapience - Principles and Methods for Wise Systems, P 77-91, Springer-Verlag, London, England, 2008.

HUHNS, M. N., STEPHES, L. M., '**Multiagent Systems and Societies of Agents**', MIT Press Cambridge, MA, USA, 1999.

JIMÉNEZ, M., PALOMERA, R. AND COUVERTIER, I., '**Introduction to Embedded Systems- Using Microcontrollers and the MSP430**', Springer New York, Heidelberg Dordrecht, London, 2014.

KARNOUSKOS, S., LEITÃO, P., '**Industrial Agents - Emerging Applications Of Software Agents In Industry**', Elsevier, SL, 2012.

MARWEDEL P., '**Embedded System Design**', Springer, Netherlands, 2006.

NEBEL, W.; SCHUMACHER, G., '**Object-Oriented HardWare Modeling – Where to Apply and What are Objects?**' In: Object-Oriented Modeling, P 101-109, Kluwer Academic Publishers, Netherlands, 1996.

NOERGAARD, T., '**Embedded Systems Architecture - A Comprehensive Guide for Engineers and Programmers**', Second Edition, Elsevier, SL, 2012.

RAO, A.S; GEORGEFF, M., '**BDI Agents: from Theory to Practice**'. In Proceedings of the 1st International Conference on Multi-Agent Systems, pp. 312–319, San Francisco, CA. 1995.

RAO, A. '**AgentSpeak(L): BDI agents speak out in a logical computable language**'. In: van Hoe, R., editor. Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands. 1996.

RUSSEL, S.; NORVIG, P., '**Artificial Inteligence: A Modern Approach**'. Third Edition, Pearson Prentice-Hall, Upper Saddle River, New Jersey, USA, 2003.

SITE FIPA, FIPA, Disponível em: < http://www.fipa.org/>, Acessado em 23 de novembro de 2015.

SITE ARDUINO, Arduino, Disponível em: < https://www.arduino.cc>, Acessado em 16 de maio de 2016.

SHEHORY, O. AND ARNON S., '**Agent Oriented Software Engineering – Reflections on Architectures, Methodologies, Languages, and Frameworks**', Springer-Verlag Berlin Heidelberg. 2014.

SHOHAN, Y. , '**Agent-Oriented Programming**', Journal of Artificial Intelligence 60, pp. 51-92, Elsevier. 1993.

SOMMERVILLE, I., '**Engenharia de *software***', Pearson Prentice-Hall, São Paulo, Brasil. 2011, p.300.

Projeto DuinOS, Disponível em: < https://github.com/DuinOS/DuinOS>, Acessado em 16 de maio de 2016.

TREVENNOR, A.; '**Practical AVR Microcontrollers - Games, Gadgets, and Home Automation with the Microcontroller Used in the Arduino**', Apress, SL, 2012.

WOOLDRIDGE, M. and JENNINGS, N. R. '**Intelligent agents: theory and practice**'. The Knowledge Engineering Review 10 (2), 115-152, (S.L), 1995.

WOOLDRIDGE, M. (2002) '**An introduction to MultiAgents Systems**', Department of Computer Science, University of Liverpool, UK. 2002.