UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

PROTÓTIPO DE AMBIENTE VISUAL DE DESENVOLVIMENTO DE PROGRAMAS PARA A PLATAFORMA EMBARCADA LCP11U37

WILLIAN DE AVILLA SILVEIRA

BLUMENAU
2019

WILLIAN DE AVILLA SILVEIRA

PROTÓTIPO DE AMBIENTE VISUAL DE DESENVOLVIMENTO DE PROGRAMAS PARA A PLATAFORMA EMBARCADA LCP11U37

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Miguel A. Wisintainer - Orientador

BLUMENAU
2019

FERRAMENTA DE ENSINO DE PROGRAMAÇÃO DE HARDWARE UTILIZANDO SCRATCH

Por

WILLIAN DE AVILLA SILVEIRA

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof(a). Miguel Alexandre Wisintainer – Orientador(a), FURB
Membro:	Prof(a). Luciana Pereira de Araújo Kohler – FURB
Membro:	Prof(a). Mauro Marcelo Mattos – FURB

Blumenau, 11 de dezembro de 2019

AGRADECIMENTOS

Primeiramente agradeço a Deus pela minha vida, também por ter me dado forças para continuar com os estudos e ter me disponibilizado tudo o que tenho hoje.

Aos meus pais Zeli da Aparecida de Avila e Lorival Ortiz Mendes por todo o apoio em qualquer situação, por sempre me incentivarem a nunca desistir dos meus sonhos e fornecerem todas as condições necessárias para que eu pudesse concluir essa graduação.

Ao meu orientador Miguel Alexandre Wisintainer pelo seu grande apoio e suporte em todas as fases do projeto, sendo importante e uma peça fundamental para que eu pudesse terminar esse trabalho.

Agradeço também aos meus amigos Luiz Henrique, Rafaela Rushinck, Juana Pereira e Jeidsan Pereira, pelo grande apoio e ajuda em todas as disciplinas principalmente na construção deste trabalho.

Por fim, agradeço à empresa Coridium Corp pela parceria, donos da ideia, a qual disponibilizaram todo os materiais e auxílio possível para o desenvolvimento do TCC.

“Talvez não tenha conseguido fazer o melhor, mas lutei para que o melhor fosse feito. Não sou o que deveria ser, mas Graças a Deus, não sou o que era antes”.

Marthin Luther King

RESUMO

O presente trabalho descreve o protótipo de uma ferramenta baseado em Blockly para a construção de uma interface gráfica, por meio de blocos, para a geração de códigos em Basic. Para desenvolver o trabalho foram levantados, primeiro, os requisitos funcionais e não funcionais necessários para então criar a especificação do trabalho. Esta é uma ferramenta que utiliza a linguagem Blockly e foi desenvolvida para processadores ARM. Estes interpretam a linguagem Basic, usando a ferramenta BASICTools para a compilação do código gerado, utilizando a plataforma embarcada LPC11u37 para executar o código. Para a implementação da interface visual foi utilizada a biblioteca Blockly, que disponibiliza todas as funcionalidades para poder começar a manipular a interface dos blocos. Seu desenvolvimento foi realizado a partir das linguagens JavaScript, CSS e HTML. Para transformar o código gerado em binário foi utilizado o compilar BASICTools, enquanto a API Electron foi utilizada para transformar a aplicação web para desktop, ainda utilizando o navegador de forma *off-line*. O resultado foi uma aplicação capaz de traduzir montagem de blocos complexas para Basic e então abrir o compilador com o código gerado.

Palavras-chave: ARMBasic. BASICTools. Blockly. Electron. Plataforma embarcada. LPC11u37. Compilador. Blocos. Interface. Visual.

ABSTRACT

The present work describes the prototype of a Blockly-based tool for building a graphical interface, through blocks, for Basic code generation. To develop the work were first raised the functional and non-functional requirements needed to then create the job specification. This is a tool that uses Blockly language and was developed for ARM processors. These interpret the Basic language, using the BASICTools tool to compile the generated code, using the LPC11u37 embedded platform to execute the code. For the implementation of the visual interface it was used the library Blockly, which provides all the features to be able to start manipulating the interface of blocks. Its development was based on the JavaScript, CSS and HTML languages. To transform the generated code into binary, the BASICTools compile was used, while the Electron API was used to transform the web application to desktop, however, still using the browser off-line. The result was an application capable of translating complex block assembly into Basic and then opening the compiler with the generated code.

Key-words: ARMBasic. BASICTools. Blockly. Electron. Board. LPC11u37. Compiler. Block. Interface. Visual.

LISTA DE FIGURAS

Figura 1 - Exemplo de programa construído utilizando a interface gráfica da ferramenta..... 17
Figura 2 - Algoritmo criado utilizando o blockly..... 17
Figura 3 - Código em JavaScript do algoritmo da Figura 2..... 18
Figura 4 – Estrutura de arquivos dos programas em Blockly..... 18
Figura 5 – Interface `BlockFactory`..... 19
Figura 6 – Interface `Code` do Blockly..... 20
Figura 7 – Estrutura de arquivos do Electron..... 22
Figura 8 – Placa LPC11U37..... 26
Figura 9 – Diagrama pinos da placa LPC11U37..... 27
Figura 10 – Estrutura interna placa LCP11U37..... 27
Figura 11 – Interface BASICtools..... 29
Figura 12 – Interface programável do BrainPad..... 30
Figura 13 – Aplicação Blynk..... 31
Figura 14 – Ambiente de programação gráfica do Visuino..... 32
Figura 15 - Diagrama de casos de uso..... 34
Figura 16 – Diagrama de sequência, criar programa no Blockly..... 38
Figura 17 – Diagrama de sequência, geração de código..... 39
Figura 18 – Diagrama de sequência, excluir todos os blocos..... 40
Figura 19 – Diagrama de sequência, salvar programa..... 40
Figura 20 – Diagrama de sequência, carregar programa..... 41
Figura 21 – Construção do bloco `IFELSE`..... 43
Figura 22 – Bloco `IFELSE`..... 44
Figura 23 – Estrutura do bloco `controls_repeat`..... 46
Figura 24 – Bloco `controls_repeat`..... 46
Figura 25 – Estrutura do bloco `Function`..... 47
Figura 26 – Bloco `procedures_defnoreturn`..... 48
Figura 27 – Hardwares conectados..... 52
Figura 28 - Interface do protótipo..... 53
Figura 29 – Categoria Logic na Toolbox..... 54
Figura 30 – Instruções adicionadas..... 55

Figura 31 – Teste do exemplo da Figura 30 55

LISTA DE QUADROS

Quadro 1 – Código bloco `Math` 19
Quadro 2 – Exemplo de código fonte das palavras reservadas do Python 21
Quadro 3 – Código fonte, sequência de operadores. 21
Quadro 4 - Comando para instalar o pré-build do Electron 23
Quadro 5 - Comando para instalar o Electron 23
Quadro 6 - Comando para gerar executável 23
Quadro 7 - Instalar Prompt no Electron 24
Quadro 8 – Exemplo de Linguagem Basic. 25
Quadro 9 – Palavras reservadas do Basic 25
Quadro 10 – Caso de uso UC01 35
Quadro 11 – Caso de uso UC02 35
Quadro 12 - Caso de uso UC03 36
Quadro 13 - Caso de uso UC04 36
Quadro 14 - Caso de uso UC05 37
Quadro 15 - Caso de uso UC06 38
Quadro 16 – Estrutura HTML da interface gráfica 42
Quadro 17 – Blocos e categorias atuais 43
Quadro 18 – Código do bloco `IFESLE` 44
Quadro 19 – Incluir o bloco na interface gráfica 45
Quadro 20 – Código do gerador de Basic do bloco `controls_if` 45
Quadro 21 – Código gerado pelo bloco `IFELSE` 45
Quadro 22 – Código do bloco `controls_repeat` 46
Quadro 23 – Código do gerador de Basic do bloco `controls_repeat` 47
Quadro 24 – Código gerado pelo bloco `controls_repeat` 47
Quadro 25 – Código do bloco `procedures_defreturn` 48
Quadro 26 - Código do gerador de Basic do bloco `procedures_defnoreturn` 49
Quadro 27 – Código gerado pelo bloco `procedures_defnoreturn` 49
Quadro 28 – Código fonte, função `finish` 49
Quadro 29 - Configuração do `package.json` Electron 50
Quadro 30 - Arquivo `Main.js`, declarações 51

Quadro 31 - Arquivo Main.js, `BrowserWindow` 51
Quadro 32 - Arquivo Main.js, carregar tela principal e Menu 51
Quadro 33 - Arquivo Main.js, carregar aplicação 52
Quadro 34 – Perguntas feitas à Coridium 56
Quadro 35 – Comparativo entre trabalhos relacionados 57

LISTA DE ABREVIATURAS E SIGLAS

A/D – Analog/Digital

API – Application Programming Interface

BASIC – Beginner's All-purpose Symbolic Instruction Code

CMD – Command Prompt

CMS – Content Management System

CSS – Cascading Style Sheet

EEPROM – Electrically Erasable Programmable Read-Only Memory

E/S – Entrada/Saída

GND – Ground

HTML – Hypertext Markup Language

I2C – Inter Integrated Circuit

IO – Input/Output

IoT – Internet of Things

LED – Light Emitting Diode

PWM – Pulse-Width Modulation

SPI – Serial Peripheral Interface

SSP – Synchronous Serial Port

TTL – Transistor Transistor Logic

UART – Universal Asynchronous Receiver Transmitter

UML – Unified Modeling Language

VPLs – Visual Programming Languages

W3C – World Wide Web Consortium

SUMÁRIO

1 INTRODUÇÃO 13
1.1 OBJETIVOS 14
1.2 ESTRUTURA 14
2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 INTERNET DAS COISAS 15
2.2 BLOCKLY 16
2.3 ELECTRON 22
2.4 LINGUAGEM BASIC 24
2.5 PLACA LPC11U37 25
2.5.1 Arquitetura 26
2.6 BASICTOOLS 28
2.7 TRABALHOS CORRELATOS 29
2.7.1 BrainPad 29
2.7.2 Blink 30
2.7.3 Visuino 31
3 DESENVOLVIMENTO 33
3.1 REQUISITOS 33
3.2 ESPECIFICAÇÃO 33
3.2.1 Casos de uso 33
3.2.2 Diagrama de sequência 38
3.3 IMPLEMENTAÇÃO 41
3.3.1 Técnicas e ferramentas utilizadas 41
3.3.2 Operacionalidade da implementação 52
3.4 ANÁLISE DOS RESULTADOS 56
3.4.1 Teste de usabilidade do protótipo 56
3.4.2 Comparativo com trabalhos correlatos 57
4 CONCLUSÕES 59
4.1 EXTENSÕES 59
REFERÊNCIAS 60

1 INTRODUÇÃO

Godinho et al. (2017, p.2) afirma que é cada vez mais comum o uso de tecnologias computacionais no dia a dia dos brasileiros, nas mais diversas atividades e finalidades, tais como no acesso a informação, na comunicação, no entretenimento ou em ferramentas de trabalho. Entretanto, a maioria das pessoas são, ainda, apenas usuários de *softwares* e aplicativos por meio de seus computadores e celulares e, possuem habilidades para utilizá-los, mas não compreendem como são desenvolvidos e como funcionam.

É visível a grande evolução da informática no decorrer dos últimos anos e a tendência é que esta área evolua ainda mais, necessitando de profissionais qualificados que possam desempenhar um bom trabalho (BEZERRA; DIAS, 2014). Porém, sabe-se que esta é uma área que exige bastante esforço, pelo seu grau de dificuldade, principalmente no que se diz respeito à lógica de programação, que é um dos requisitos fundamentais nos cursos de computação (PEREIRA; RAPKIEWICZ, 2004).

O Google Developers (2019), a fim de diminuir essa dificuldade, disponibilizou a biblioteca Blockly para que fosse possível a construção de VPLs. A biblioteca possui código aberto para programação de interfaces gráficas. Sendo um projeto desenvolvido pela equipe da Google, ela é baseada no Scratch[1] e utiliza blocos visuais que se vinculam para facilitar a escrita de código e podem gerar código em JavaScript, Dart, Python ou PHP. Também pode ser personalizado para gerar código em qualquer linguagem de programação textual.

Para Branco (2008, p.1), o nome “compilador” faz referência ao processo de composição de um programa pela reunião de várias rotinas de bibliotecas, a tradução de linguagem abstrata para linguagem de baixo nível que é executada pelo compilador. O compilador tem duas tarefas básicas: Análise, que examina, verifica e compreende o texto de entrada (no código-fonte); Geração de código, em que o texto de saída (código de máquina) é gerado de forma correspondente ao texto de entrada (código-fonte da linguagem abstrata).

Coridium, empresa parceira, contribuiu com o desenvolvimento do protótipo aqui analisado. Ela disponibilizou os materiais necessários para que fosse possível realizar os testes, sendo a placa LPC11U37 um dos materiais. A placa possui um processador ARM, o qual faz possível a interpretação das linguagens de programação C e Basic e possui um tamanho de 1,6x0,9cm. Por ser um *hardware* e interpretar apenas linguagem de máquina foi necessária a utilização de um compilador para traduzir a linguagem Basic gerada pelo

[1] *Scratch* é uma linguagem que utiliza VLP para desenvolver um ambiente gráfico, no qual é possível montar programas através de ligações de blocos.

protótipo em linguagem de máquina e então transferir essa tradução para a placa. O compilador, que é um ambiente de programação em Basic, foi também disponibilizado pela empresa Coridium para que fosse possível realizar essa transição de dados entre o computador e a placa LPC11U37.

Com base nesses argumentos propõe-se a integração do ambiente Blockly para o desenvolvimento de um protótipo de programação visual, ao qual através de estruturas de blocos complexas, que possibilite a geração de código na linguagem Basic. O BASICTools foi utilizado para traduzir o código Basic gerado pelo protótipo e então transferir o código para a placa LPC11U37. Para que fosse possível utilizar a aplicação desenvolvida em JavaScript em Desktop foi utilizado a API Electron.

1.1 OBJETIVOS

O objetivo deste trabalho é disponibilizar um ambiente visual de desenvolvimento para a plataforma embarcada LPC11u37.

Os objetivos específicos são:

a) construir um ambiente para edição de programas baseado num dialeto de Basic;
b) desenvolver um conjunto de testes para validar o projeto;

1.2 ESTRUTURA

Este trabalho está dividido em quatro capítulos. O primeiro capítulo apresenta a introdução e os objetivos do trabalho. O segundo capítulo trata da fundamentação teórica, explicando os principais conceitos e técnicas utilizados no trabalho. O terceiro capítulo contempla o desenvolvimento do protótipo, no qual são descritos a arquitetura do trabalho por meio de diagramas, detalhamento da implementação e os resultados obtidos. Por fim, o quarto capítulo apresenta as conclusões, assim como sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo está organizado em sete seções. A seção 2.1 complementa o conceito de IoT. A seção 2.2 apresenta o ambiente de programação visual Blokly. A seção 2.3 apresenta o Electron e os conceitos. A seção 2.4 apresenta a linguagem Basic para processadores ARM's e suas principais funcionalidades. A seção 2.5 apresenta a placa LPC11u37 e sua estrutura. A seção 2.6 apresenta o compilar BASICTools. Por fim, a seção 2.7 apresenta os trabalhos correlatos.

2.1 INTERNET DAS COISAS

Até o presente momento pode-se considerar que a principal e mais emergente e inovadora forma digital de comunicação entre humanos é a Internet (COLÉGIO WEB, 2013). Decorrente disso, a Internet das Coisas ou *Internet of Things* (IoT) eclode como a evolução da internet e de um novo paradigma social, cultural e principalmente tecnológico. A IoT revolucionará a interação da sociedade com o todo, fazendo com que os limites existentes relativos a esses ambientes se tornem cada vez mais tênues (ZABADAL; LISBOA MURTA DE CASTRO, 2019).

A Internet das Coisas é uma rede de objetos físicos, veículos, prédios e outros que possuem tecnologia embarcada, sensores e conexão com a rede e é capaz de coletar e transmitir dados. A próxima geração da Internet será a Internet das Coisas (IoT), que pode ser conceituada como uma rede mundial de objetos interconectados. Neste novo modelo, qualquer objeto que possua uma identificação exclusiva poderá se juntar à rede conhecida como a Internet.

A conexão com a rede mundial de computadores viabilizará, primeiro, controlar remotamente os objetos e, segundo, permitir que os próprios objetos sejam acessados como provedores de serviços. A Internet das Coisas revolucionará os modelos de negócios e a interação da sociedade com o meio ambiente. Por meio de objetos físicos e virtuais, em que esses limites se tornam cada vez mais tênues (LACERDA; LIMA-MARQUES, 2015).

Entende-se que tal tecnologia de conexão pode ser de diferentes tipos (fios de cobre, fibra ótica, ondas eletromagnéticas ou outras). Peterson (2011) definiu que a principal característica das Redes de Computadores é a sua generalidade, isto é, elas são construídas sobre dispositivos de propósito geral e não são otimizadas para fins específicos tais como as redes de telefonia e TV.

Os ambientes de Internet das Coisas são caracterizados por muita heterogeneidade. Por conta da especificidade de cada solução pode variar dentro de alguns pilares como: alcance de

comunicação, taxa de transmissão, consumo de energia e custo. Tratar essa heterogeneidade é o objetivo de uma plataforma de Internet das Coisas. Essa heterogeneidade gera um grande desafio para a criação e definição de padrões de comunicação para as coisas na internet, similar ao que já existe na *World Wide Web*. Ainda é difícil falar em padrões dentro desse universo, evidenciando a importância das plataformas, que nada mais são do que uma infraestrutura que visa facilitar a interação dos dispositivos (*hardware*, as coisas) com as aplicações (*software*) (DORNELAS; CAMPELLO, 2017).

2.2 BLOCKLY

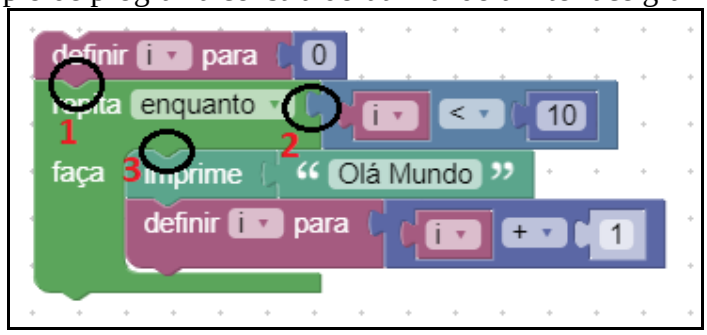
O Blockly é um ambiente de programação visual que funciona na web e tem código aberto, possibilitando o uso em ferramentas próprias de cada desenvolvedor. A biblioteca Blockly serve para adicionar um editor de código visual em aplicações web e Android. O editor Blockly utiliza blocos gráficos conectáveis para representar os conceitos de código como repetição, condição, *loops*, entre outros. Permitindo, assim, que os usuários usem de princípios de programação sem ter de se preocupar com erros de sintaxe (Blockly API, 2019).

O programa possui código aberto, o que permite que seja utilizado em aplicações próprias dos usuários e, também permite que os programas criados em linguagem visual sejam exportados para linguagem de programação comum, tornando a transição para programação baseada em texto. É uma biblioteca de JavaScript, sem dependências do lado do servidor, altamente personalizável, compatível com todos os principais navegadores: Chrome, Firefox, Safari, Opera e IE (Blockly Google Developers, 2019).

A interface da aplicação foi desenvolvida levando em conta que deveria ser simples e intuitiva, para que usuários de diferentes níveis de experiência conseguissem utilizar. A biblioteca Blockly foi utilizada de forma que as instruções disponíveis no ambiente fossem apresentadas por meio de blocos conectáveis (Heinen et. al, 2015).

A Figura 1 demonstra um programa construído utilizando a interface da ferramenta. O programa em questão tem como objetivo apresentar a mensagem “Olá Mundo” 10 vezes. As conexões destacadas relacionam-se com: (1) instruções anteriores e subsequentes: (2) blocos que representam valores ou o retorno de uma função: e (3) instruções executadas dentro do escopo do bloco atual.

Figura 1 - Exemplo de programa construído utilizando a interface gráfica da ferramenta

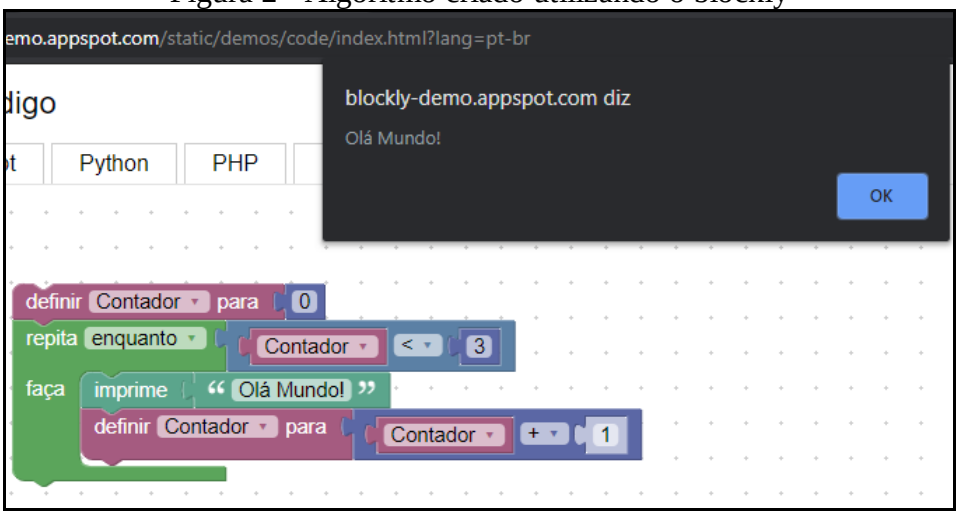


Fonte: Elaborado pelo autor.

De acordo com a Google Developers (2019), da perspectiva do usuário, o Blockly é uma maneira visual intuitiva de criar código. Para o ponto de vista de um desenvolvedor, o Blockly é uma interface pronta para criar uma linguagem visual que traduz o código gerado pelo usuário sintaticamente correto. O editor Blockly consiste em uma *Toolbox*[2] para armazenar tipos de blocos e uma *Workspace*[3] para organizar os blocos. O editor também permite que o usuário crie aplicativos utilizando o Blockly e crie blocos personalizados para serem codificados e adicionados às suas caixas de ferramentas.

Na Figura 2 é apresentado um exemplo simples de um algoritmo feito no Blockly, que consiste na definição de uma variável com o nome de “contador”, a qual é inicializada com o valor de 1. Em seguida é adicionado um laço de repetição que irá repetir até que a variável “contador” chegue ao valor 3. Em cada repetição, o algoritmo irá emitir um alerta com a frase “Olá Mundo!”, e irá incrementar ao “contador” mais 1. Totalizando assim 3 mensagens de alerta na tela.

Figura 2 - Algoritmo criado utilizando o blockly



Fonte: Elaborado pelo autor.

[2] *Toolbox* ou caixa de ferramenta é onde são armazenados os blocos, separando-os por categorias (*Logic*, *loop*, *text*, entre outros).
[3] *Workspace* ou área de trabalho é o espaço onde a aplicação com blocos é construída.

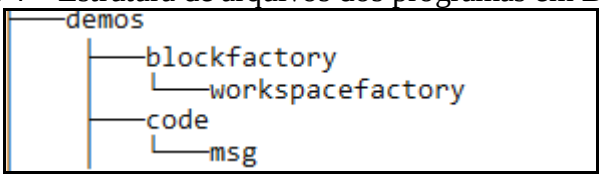
Assim como foi citado anteriormente, após criar um algoritmo ou programa utilizando o Blockly, é possível converter o mesmo em código. Assim na Figura 3 é demonstrado o algoritmo na versão JavaScript em texto.

Figura 3 - Código em JavaScript do algoritmo da Figura 2

Fonte: Elaborado pelo autor.

O Blockly tem uma estrutura de arquivos subdividida entre programas. É apresentado na Figura 4 um exemplo de algumas funcionalidades existentes, além da utilizada para a construção desse projeto. Na pasta `demos` encontra-se todas as extensões possíveis, que podem ser utilizadas para customização dos blocos e para montar os blocos a partir de códigos escritos em JavaScript ou em JSON. Também é possível incluir bibliotecas externas como Angular JS para personalizar a interface visual.

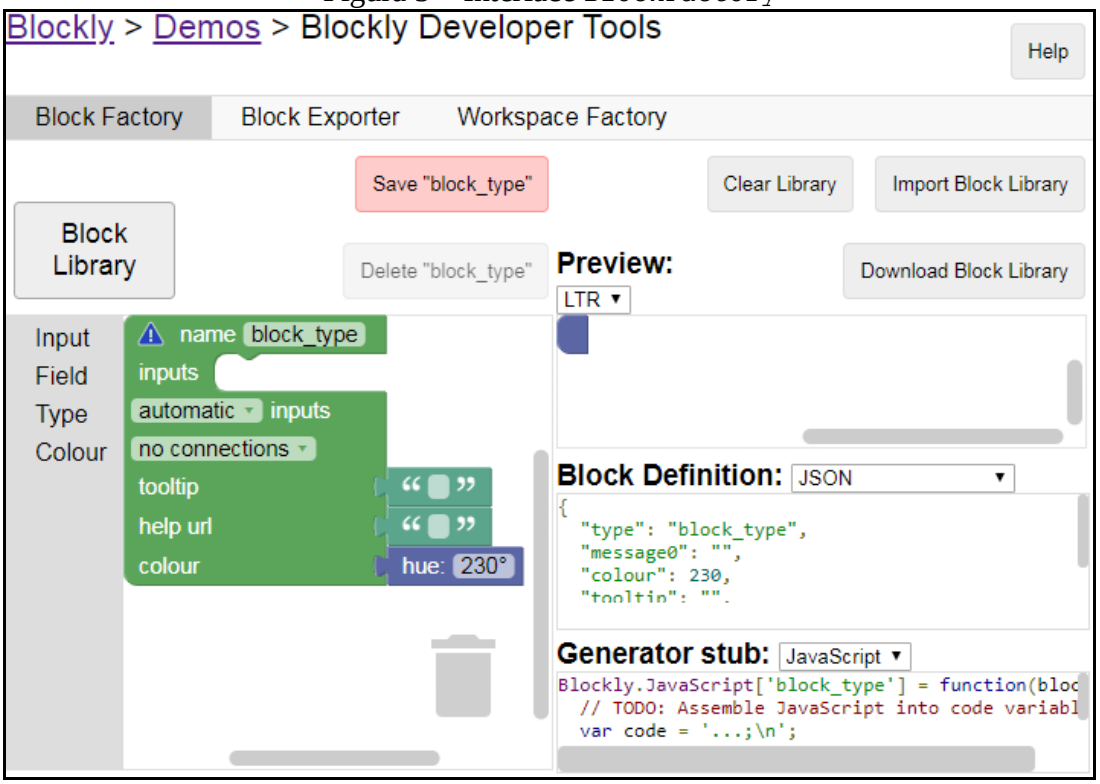
Figura 4 – Estrutura de arquivos dos programas em Blockly



Fonte: Elaborado pelo autor.

Utilizando o `BlockFactory` é possível criar blocos a partir da manipulação dos blocos, ou por meio das linguagens JSON e JavaScript. Na Figura 5 é apresentada a interface em que é realizada a construção dos blocos e na qual é possível identificar elementos para a criação de blocos a partir de blocos definidos. Também pode ser observado que há uma aba "*preview*" para avaliação de como ficará o bloco após montá-lo usando as linguagens descritas. Após esses passos é gerado um código básico que pode ser escolhido entre JSON ou JavaScript e será automaticamente formatado para ser posto no protótipo.

Figura 5 – Interface `BlockFactory`



Fonte: Elaborado pelo autor.

A construção do bloco é realizada por meio das linguagens JSON ou JavaScript, sendo possível em JavaScript adicionar funções, métodos e atributos extras para serem utilizados em outros trechos do código. Um exemplo é apresentado no Quadro 1, onde é criado o bloco `Math`, o que possui como entrada apenas um dado numérico.

Quadro 1 – Código bloco `Math`

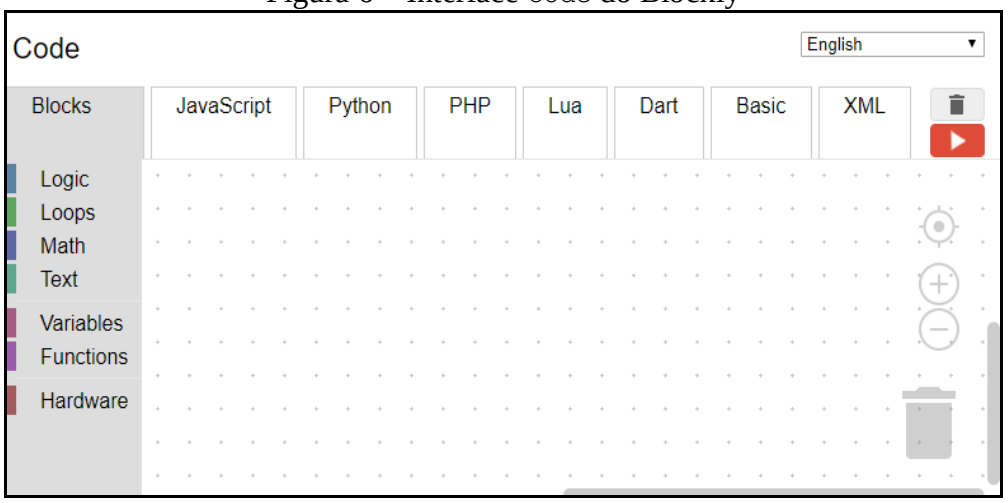
```
{
  "type": "math_number",
  "message0": "%1",
  "args0": [{
    "type": "field_number",
    "name": "NUM",
    "value": 0
  }],
  "output": "Number",
  "helpUrl": "%{BKY_MATH_NUMBER_HELPURL}",
  "style": "math_blocks",
  "tooltip": "%{BKY_MATH_NUMBER_TOOLTIP}",
  "extensions": ["parent_tooltip_when_inline"]
},
```

Fonte: Elaborado pelo autor.

A Figura 6 apresenta o *Code*, o qual serviu de base para a criação do projeto. Ele possui uma interface em que é possível alterar as linguagens existentes no Blockly. Atualmente o Blockly possui sete traduções de códigos, sendo elas: os próprios blocos,

JavaScript, Python, PHP, Lua, Dart e XML. É possível ver o código gerado pelo Blockly em todas essas linguagens.

Figura 6 – Interface `Code` do Blockly



Fonte: Elaborado pelo autor.

As demais funcionalidades existentes são para auxiliar na construção dos blocos, como redirecionamento, tamanho e cores. Também é possível alterar o *Workspace* ou o *toolbox*. Os arquivos também possuem suporte à *Storage* do navegador, sendo possível salvar o estado dos blocos montados de modo que mesmo após fechar o navegador os blocos salvos são mantidos em *Cache*. O Blockly também tem suporte para a criação de gráficos a partir da construção de programas em blocos.

O Quadro 2 apresenta a declaração das palavras reservadas da linguagem, nele é definido todas as palavras que a linguagem utiliza para uso interno e interpretação de dados. Cada linguagem possui uma lista com todas as palavras reservadas.

Quadro 2 – Exemplo de código fonte das palavras reservadas do Python

```
Blockly.Python.addReservedWords(
    // import keyword
    // print(','.join(sorted(keyword.kwlist)))
    // https://docs.python.org/3/reference/lexical_analysis.html#keywords
    // https://docs.python.org/2/reference/lexical_analysis.html#keywords
    'False,None,True,and,as,assert,break,class,continue,def,del,elif,else,' +
    'except,exec,finally,for,from,global,if,import,in,is,lambda,nonlocal,not,' +
    'or,pass,print,raise,return,try,while,with,yield,' +
    // https://docs.python.org/3/library/constants.html
    // https://docs.python.org/2/library/constants.html
    'NotImplemented,Ellipsis,__debug__,quit,exit,copyright,license,credits,' +
    // >>> print(','.join(sorted(dir(__builtins__))))
    // https://docs.python.org/3/library/functions.html
    // https://docs.python.org/2/library/functions.html
    'ArithmeticError,AssertionError,AttributeError,BaseException,' +
    'BlockingIOError,BrokenPipeError,BufferError,BytesWarning,' +
    'ChildProcessError,ConnectionAbortedError,ConnectionError,' +
    'ConnectionRefusedError,ConnectionResetError,DeprecationWarning,EOFError,' +
    'Ellipsis,EnvironmentError,Exception,FileExistsError,FileNotFoundError,' +
    'FloatingPointError,FutureWarning,GeneratorExit,IOError,ImportError,' +
    'ImportWarning,IndentationError,IndexError,InterruptedError,' +
    'IsADirectoryError,KeyError,KeyboardInterrupt,LookupError,MemoryError,' +
    'ModuleNotFoundError,NameError,NotADirectoryError,NotImplemented,' +
```

Fonte: Elaborado pelo autor.

O Quadro 3 apresenta uma lista de enumerações com todos os operadores de cada linguagem juntamente com a ordem de busca. Essas enumerações são utilizadas na geração do código para definir a precedência deles no código gerado.

Quadro 3 – Código fonte, sequência de operadores.

```
Blockly.Basic.ORDER_ATOMIC = 0;           // 0 "" ...
Blockly.Basic.ORDER_NEW = 1.1;            // new
Blockly.Basic.ORDER_MEMBER = 1.2;         // . []
Blockly.Basic.ORDER_FUNCTION_CALL = 2;    // ()
Blockly.Basic.ORDER_INCREMENT = 3;        // ++
Blockly.Basic.ORDER_DECREMENT = 3;        // --
Blockly.Basic.ORDER_BITWISE_NOT = 4.1;    // ~
Blockly.Basic.ORDER_UNARY_PLUS = 4.2;     // +
Blockly.Basic.ORDER_UNARY_NEGATION = 4.3; // -
Blockly.Basic.ORDER_LOGICAL_NOT = 4.4;    // <>
Blockly.Basic.ORDER_TYPEOF = 4.5;         // typeof
Blockly.Basic.ORDER_VOID = 4.6;           // void
Blockly.Basic.ORDER_DELETE = 4.7;         // delete
Blockly.Basic.ORDER_AWAIT = 4.8;          // await
Blockly.Basic.ORDER_EXPONENTIATION = 5.0; // **
Blockly.Basic.ORDER_MULTIPLICATION = 5.1; // *
Blockly.Basic.ORDER_DIVISION = 5.2;       // /
Blockly.Basic.ORDER_MODULUS = 5.3;        // %
Blockly.Basic.ORDER_SUBTRACTION = 6.1;    // -
Blockly.Basic.ORDER_ADDITION = 6.2;       // +
Blockly.Basic.ORDER_BITWISE_SHIFT = 7;    // << >> >>>
Blockly.Basic.ORDER_RELATIONAL = 8;       // < <= > >=
Blockly.Basic.ORDER_IN = 8;               // in
```

Fonte: Elaborado pelo autor.

2.3 ELECTRON

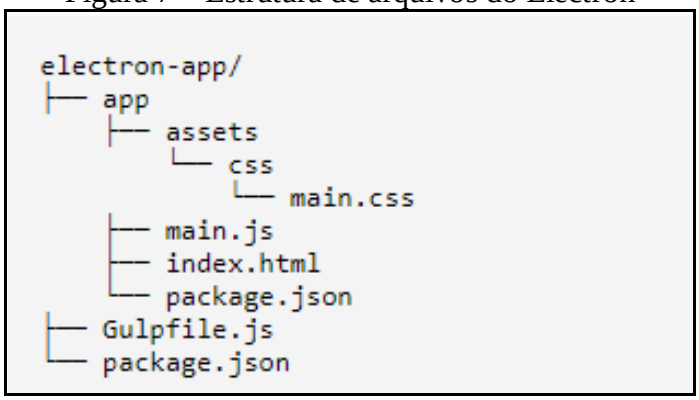
De acordo com o Electron[4] (2019), a API Electron é um *framework* para criação de aplicações nativas utilizando tecnologias web como JavaScript, HTML e CSS. O Electron é uma biblioteca de código aberto desenvolvida pelo GitHub para o desenvolvimento de aplicativos multiplataformas usando HTML, CSS e JavaScript. Aplicativos Electron podem ser empacotados para Mac, Windows e Linux.

Para o Electron (2019) aplicações eletrônicas são compostas de múltiplos processos. Existe o processo "*browser*" e vários processos "renderizador". O processo do navegador executa a lógica do aplicativo e pode iniciar vários processos do renderizador, fazendo, assim, o processo ajustar as janelas que aparecem na tela do usuário, alterando o HTML e CSS. Os processos do navegador e do renderizador podem ser executados com a integração do Node.js[5] se ativado. A maioria das APIs da Electron são escritas em C ++ ou Objective-C e depois expostas diretamente ao código do aplicativo através de ligações JavaScript.

De acordo com Danin (2019), os motivos para usar o Electron são: suporta os principais sistemas operacionais (macOS, Linux, Windows) e possui a capacidade de usar funções nativas do sistema como exibição de notificações.

A Figura 7 apresenta a estrutura de arquivos do Electron, no qual na pasta “app” contém todos os elementos relacionados ao projeto. O “main.js” é o arquivo de inicialização da aplicação, ao qual vão as configurações do tipo: tamanho da tela, posicionamento, manipular eventos do sistema, entre outros. O arquivo “index.html” consiste na especificação da tela principal e o arquivo “package.json”, localizada na raiz do projeto, é responsável pelas configurações e dependências para o ambiente de desenvolvimento.

Figura 7 – Estrutura de arquivos do Electron



Fonte: SOSA (2015).

[4] https://electronjs.org/

[5] *Node.js* é uma plataforma construída sobre o motor JavaScript do Google Chrome para construir aplicações de rede rápidas e escaláveis.

O Electron é utilizado para criar aplicações Desktop desenvolvidas em JavaScript, HTML e CSS. Como pré-requisito para a utilização é necessário a instalação do Node.js, pois o Electron utiliza elementos do Node.js para poder instalar e configurar os arquivos. Além do Node.js é preciso que seja instalado o pacote global do Electron, apresentado no Quadro 4, o qual possui uma estrutura pronta para montar a aplicação.

Quadro 4 - Comando para instalar o pré-build do Electron

```
npm install -g electron-prebuilt
```

Fonte: Elaborado pelo autor.

A utilização do Electron faz com que a aplicação construída utilizando HTML, JavaScript e CSS se torne Desktop, assim possibilitando o uso de funcionalidades para acesso e gravação de arquivos no disco do computador, livrando o uso de um servidor "back-end" e a publicação da aplicação em um servidor Apache. Para este trabalho foi utilizado como base poucas funcionalidades do Electron, sendo as principais, a portabilidade de aplicação GTML para Desktop e o uso do "dialog" do Windows.

O Electron tem como base o uso do Node.js, sendo um requisito para a instalação. Após a instalação, os arquivos da aplicação devem conter a mesma estrutura apresentada na Figura 7. A utilização do Electron é por meio da janela de comando do Windows (CMD), após estruturar a pasta da aplicação, através do CMD a pasta a qual está o projeto deve ser referenciada, utilizando o comando "cd C:\", após isso, deve-se utilizar o comando apresentado no Quadro 5.

Quadro 5 - Comando para instalar o Electron

```
npm install electron
```

Fonte: Elaborado pelo autor.

Já o Electron Packager é responsável por empacotar a aplicação com o Electron e gerar um executável; logo, para gerar o executável, é necessário que o Electron esteja instalado. Após executar o comando de instalação do Packager, para gerar o executável com ele, utiliza-se o comando "electron-packager ." ao qual o "." indica que o diretório do projeto é o mesmo em que foi definido na instalação, ou seja, o mesmo em que estamos executando o comando.

Quadro 6 - Comando para gerar executável

```
npm install electron-packager

electron-packager .
```

Fonte: Elaborado pelo autor.

O Electron disponibiliza algumas funcionalidades, mas para isso é preciso realizar o mesmo procedimento manual. No Quadro 7 apresenta-se o "prompt", usado para gerar alertas do Windows na aplicação, podendo, assim, usar as mensagens de erro e confirmações do Sistema operacional.

Quadro 7 - Instalar Prompt no Electron

```
npm install electron-prompt –save
```

Fonte: Elaborado pelo autor.

2.4 LINGUAGEM BASIC

De acordo com Sorokanich (2014), a linguagem BASIC foi desenvolvida por Thomas E. Kurtz e John G. Kemeny, membros do departamento de matemática de Dartmouth, em 1963:

> O BASIC foi desenvolvido com o intuito de ser uma linguagem fácil e interativa, de modo a possibilitar seu uso por cientistas da computação, que rapidamente pudessem criar programas e executá-los. (Sorokanich, 2014).

Segundo Morimoto (2005, p.1), os programas em BASIC são construídos por meio da combinação de comandos simples, baseados em palavras do Inglês e rodam, linha a linha, à medida que são "traduzidos" para linguagem de máquina pelo interpretador. Ainda segundo Morimoto (2005, p.1) os compiladores BASIC atuais são bem mais rápidos e mais flexíveis que os desta primeira geração e, apesar de ainda não serem tão rápidos quanto programas em C, já são capazes de fazer praticamente tudo que é possível em outras linguagens. Um exemplo de linguagem popular atualmente que é baseada no BASIC é o Visual Basic da Microsoft sendo o principal benefício do Basic poder criar aplicações de baixo e médio porte com relativa simplicidade e rapidez, enquanto em outras linguagens, como o C/C++ e Java, o processo é mais longo.

Segundo a Coridium, um dos objetivos do ARMbasic é ser uma linguagem simples e fácil de usar, mas ainda assim ser uma ferramenta para controlar o *hardware*. Por esse motivo, um subconjunto do BASIC foi escolhido com extensões para controle de *hardware*. No ARMbasic o tipo de dados padrão é 32 bits (SIGLE e INTEGER) e também suporta matrizes de SIGLE, INTEGERS e STRINGs de bytes terminados por zero e números de ponto flutuante IEEE 754 de 32 bits. Somente matrizes de dimensão única são suportadas.

O Quadro 8 apresenta um trecho de um código em BASIC adaptado para plataformas embarcadas. O exemplo é de um programa feito para acessar a posição de um vetor, na primeira linha é declarado o vetor com dez posições do tipo inteiro, então no método “main”, na posição 5 do vetor é inserido o valor de número 5. Logo após, essa posição 5 é apresentada na tela com o comando “PRINT”.

Quadro 8 – Exemplo de Linguagem Basic.

```
DIM vetor(10) as integer

main:
  vetor(5) = 5
  PRINT vetor(5)
END
```

Fonte: Elaborado pelo autor.

Cada linguagem de programação possui palavras que não podem ser utilizadas para declaração de variáveis, ou seja, palavras reservadas pelo sistema. No Quadro 9 são apresentadas algumas das palavras reservadas usadas pelo ARMBasic e, também se deve pôr como palavra reservada o próprio Blockly, para que não seja utilizado.

Quadro 9 – Palavras reservadas do Basic

```
Blockly.Basic.addReservedWords(
   //Basic
   //https://www.coridium.us/ARMhelp/index.htm
   //->The Language -> Alphabetical KEYWORDs
   'abs','ad','addressof','and','as','_asm_','baud','byref','byval' +
   'call','case','chr','clear','const','debugin','dim','dir','do','loop'+
   'downto','else','elseif','end','endfunction','endif','endselect'+
'endsub','exit','fileopen','filereaddir','filerreadbyte','filewritebyte'+
   'fileof','fileclose','for','fread','gosub','goto','hex','high'+
   'if','then','in','input','integer','interrupt','io','left','len'+
   'list','loop','low','mail','main','mod','next','not','on'+
   'or','out','output','print','printf','return','right','rnd'+
   'run','rxd','select','single','sleep','sprintf','slep','stop'+
   'str','strcomp','string','sub','then','timer','to','txd','udpin'+
   'udpout','until','val','wait','waitmicro','while','write','xor');
```

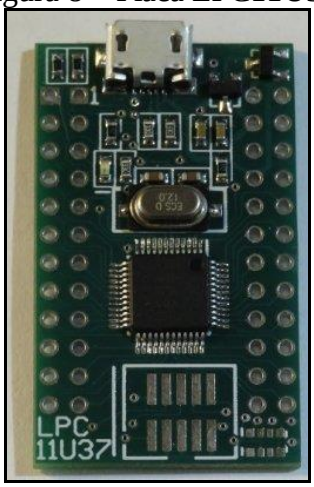
Fonte: Coridium (2019).

2.5 PLACA LPC11U37

Possuindo um tamanho de 1.6x0.9cm, sua arquitetura é composta por uma CPU ARM M0, que trabalha na frequência de 50 MHz e contém 40 pinos que funcionam a 5 Volts, tendo 8 conversores de A/D de 10 Bits. Sobre a placa há um módulo de memória para armazenar dados de usuários e códigos enviados para execução, as quais 8K são especificamente para armazenar dados. Tem um espaço de 128K para armazenar o código fonte, a placa abriga um módulo EEPROM de 4K, ao qual armazena dados de forma não volátil e mantém suas informações mesmo após desligar o equipamento.

Segundo a empresa Coridium (2019), o Parallax introduziu o BASIC Stamp que por algum tempo foi a ferramenta de laboratório para o controle simples do microprocessador. Esse *design* nunca foi atualizado. A placa é apresentada na Figura 8, onde é possível identificar seu design e sua forma, nela é possível identificar que a sua alimentação de força é por meio de um conector micro USB, ao qual é ligada diretamente ao computador.

Figura 8 – Placa LPC11U37



Fonte: Elaborada pelo autor.

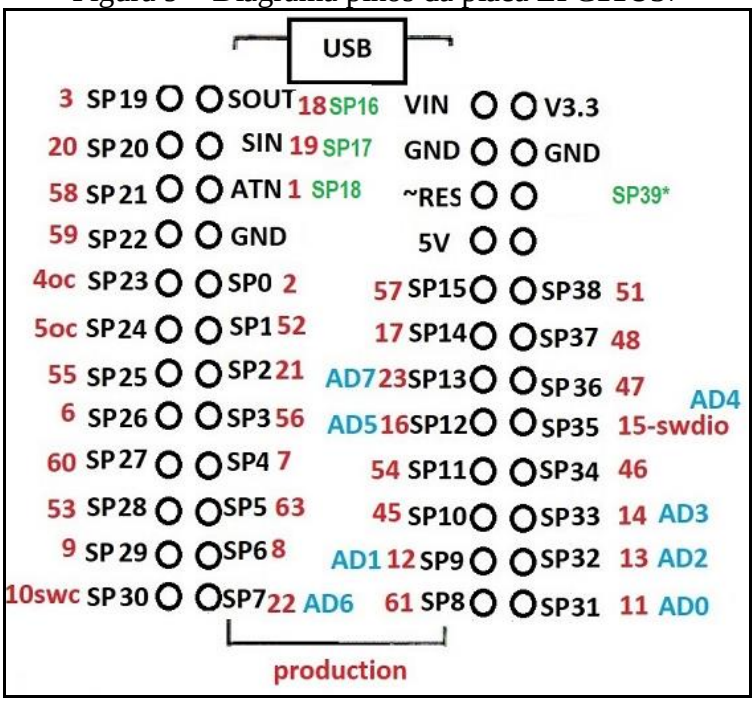
Para a Coridium (2019) esta placa substitui a conexão de depuração RS-232 por USB. A placa também pode ser programada em BASIC ou C utilizando o ambiente BASICTools, disponibilizado pela empresa. Também contém bibliotecas que fornecem suporte para funções matemáticas e acesso ao *hardware*.

2.5.1 Arquitetura

De acordo com a empresa supracitada, a placa LPC11U37 possui um conjunto aplicado de pinos em relação ao antigo Parallax BASIC. A placa pode ser usada com sinais TTL de 5V e possui um total de 40 pinos de E/S disponíveis. Os programas BASIC ou C podem ser baixados usando o conector USB. As ferramentas MakeItC e BASIC usam a conexão serial USB da placa, a qual permite que o programa carregue programas mBed fazendo um curto-circuito nos pinos ATN e GND, que faz com que a placa se apresente para o sistema operacional como uma unidade de disco, permitindo sobrepor o arquivo “firmware.bin”. A placa possui versão com um conector JTAG/SW opcional e com um depurador JTAG/SW que pode ser conectado a outras CPUs ARM. O LPC11U37 suporta várias funções dedicadas, dividido em uma UART, dois SSP/SPI, uma I2C, I2S, dois PWMs multicanais.

A Figura 9 apresenta um diagrama dos pinos, com o conector USB mostrado na parte superior. Os pinos NXP P0.1-31 (mostrados como 1-31) e P1.0-31 (mostrado como 32-63) são associados a vermelho na figura e por conveniência, podem ser referidos como SP0-39, usando as definições em STAMP_PINS.bas e STAMP_PINS.h (mostradas em preto e verde abaixo). Além disso, existe um conversor A/D com um multiplexador analógico de 8 entradas. Depois que os pinos são ligados e, quando um programa solicita uma entrada AD, eles são convertidos em analógicos (essas entradas estão representadas na cor azul na figura).

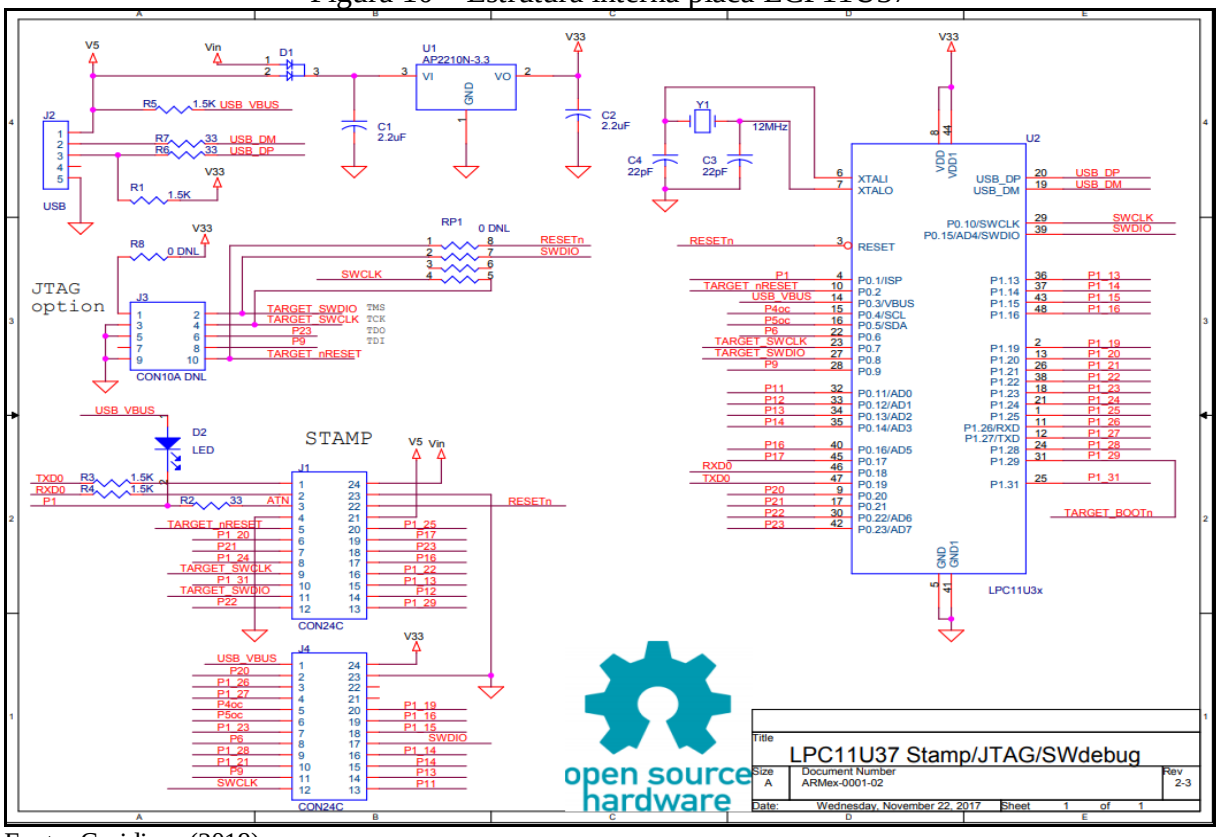
Figura 9 – Diagrama pinos da placa LPC11U37



Fonte: Coridium (2019).

Na Figura 10 é possível identificar a estrutura interna da placa apresentando cada componente e suas devidas ligações.

Figura 10 – Estrutura interna placa LCP11U37



Fonte: Coridium (2019).

2.6 BASICTOOLS

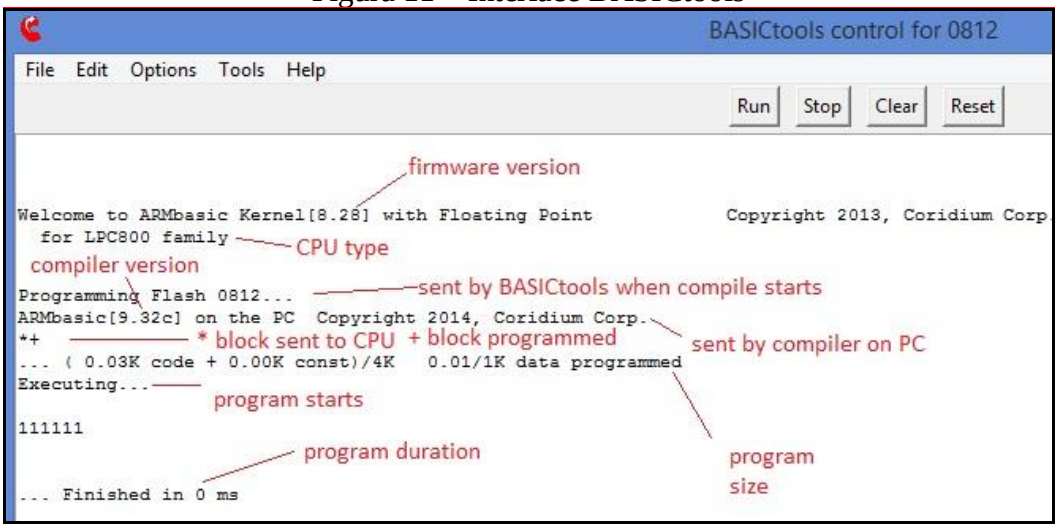
Segundo a empresa Coridium (2019) o ARMbasic é um compilador BASIC de 32 bits para processadores ARM. Foi criado com uma alternativa portátil e depuradora de *hardware*, mas rapidamente se transformou em uma poderosa ferramenta para controladores programáveis. Inclui suporte para operações assíncronas em série, I2C, SPI, PWM, *timer* e contador. É executado em CPUs ARM do NXP, variando entre 50 MHz 32KB BASICchip e 100 MHz 256KB SuperPRO.

Os produtos Coridium usam um compilador BASIC que é executado no computador. Em parceria com a empresa supracitada, foi disponibilizado por esta o embiente BASICTools, que inclui um emulador de terminal e um ambiente de desenvolvimento projetado especificamente para as placas da família PRO, BASICchip, ARMmite e Ethernet. Além disso, vários arquivos de ajuda e documentos sobre o ARMbasic foram instalados na máquina.

Para a empresa Coridium (2019), o ARMbasic depende de uma conexão USB ao PC para programação. Com os aplicativos de destino incluem-se funções de controle e um conjunto de rotinas de *hardware*. Além de ter uma sintaxe o mais compatível possível com o MS-VisualBASIC, o ARMbasic apresenta vários novos recursos, como rotinas específicas de *hardware*, suporte a *strings*, ponteiros limitados e muitos outros. O ARMbasic é escrito em ANSI-C e compilado com o GCC.

Na Figura 11, é apresentado a interface do BASICtools, em que é possivel identificar os principais elementos da tela, como a versão do *firmware* utilizado pelo processador ARM, e também verificar o tipo da placa. Nas linhas restantes está apresentada a versão do compilador, o código enviado para a memória *flash*, o código enviado para a placa, o tamanho ocupado pelo programa enviado e a duração da execução. Na tela também pode ser visto botões que são utilizados para rodar o programa, pará-lo, limpar a tela ou reiniciar a aplicação.

Figura 11 – Interface BASICtools



Fonte: Coridium (2019).

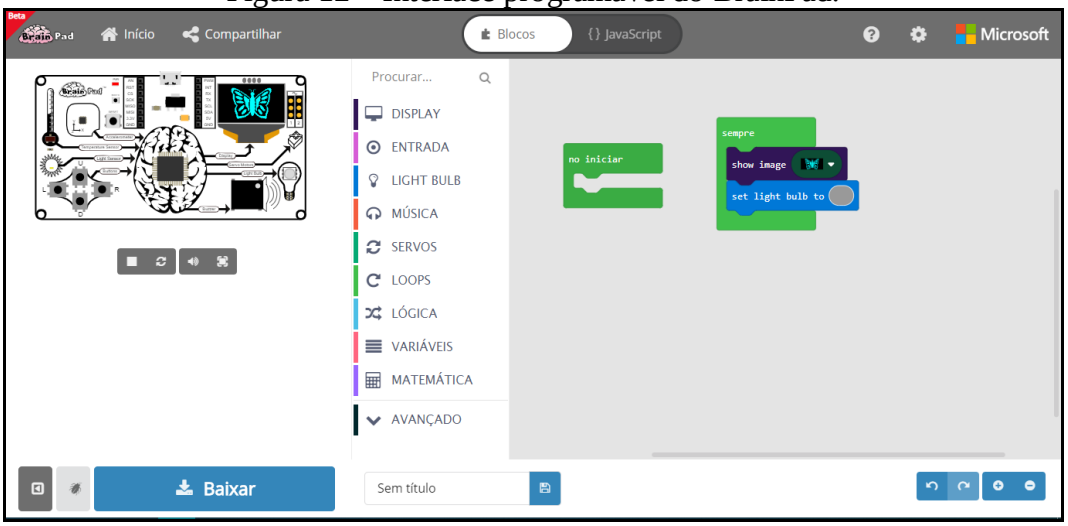
2.7 TRABALHOS CORRELATOS

São apresentados três trabalhos correlatos, que possuem características semelhantes à proposta deste trabalho. A seção 2.7.1 apresenta o Brainpad (2019), um minicomputador educacional para programação de jogos, robôs e plataforma embarcada, desenvolvida pela empresa GHI Electronics (2019). A seção 2.7.2 apresenta Blynk (BAIBORODIN, 2009), uma aplicação móvel integrada a um dispositivo eletrônico programável em um ambiente gráfico sem a utilização de muitas linhas de código. A seção 2.7.3 discorre sobre o Visuino (VISUINO, 2019), um ambiente desenvolvido para programação gráfica utilizando Arduino.

2.7.1 BrainPad

O BrainPad (GHI ELETRONICS, 2018) é um dispositivo eletrônico, capaz de simular diversas funcionalidades contendo diversos sensores. A interface foi construída utilizando uma biblioteca da Microsoft que implementa a linguagem Scratch, o que facilita a sua utilização no ambiente escolar.

Figura 12 – Interface programável do BrainPad.



Fonte: BrainPad (2019).

Conforme BraindPad (2019), por meio do *software*, o usuário pode construir diversas aplicações que podem ser alteradas via linguagem JavaScript ou utilizando os blocos da linguagem do Scratch. No Brainpad há diversos sensores e cada um deles está associado a um bloco no *software* e um conjunto de instruções. O ambiente permite a opção para alteração de código via JavaScript.

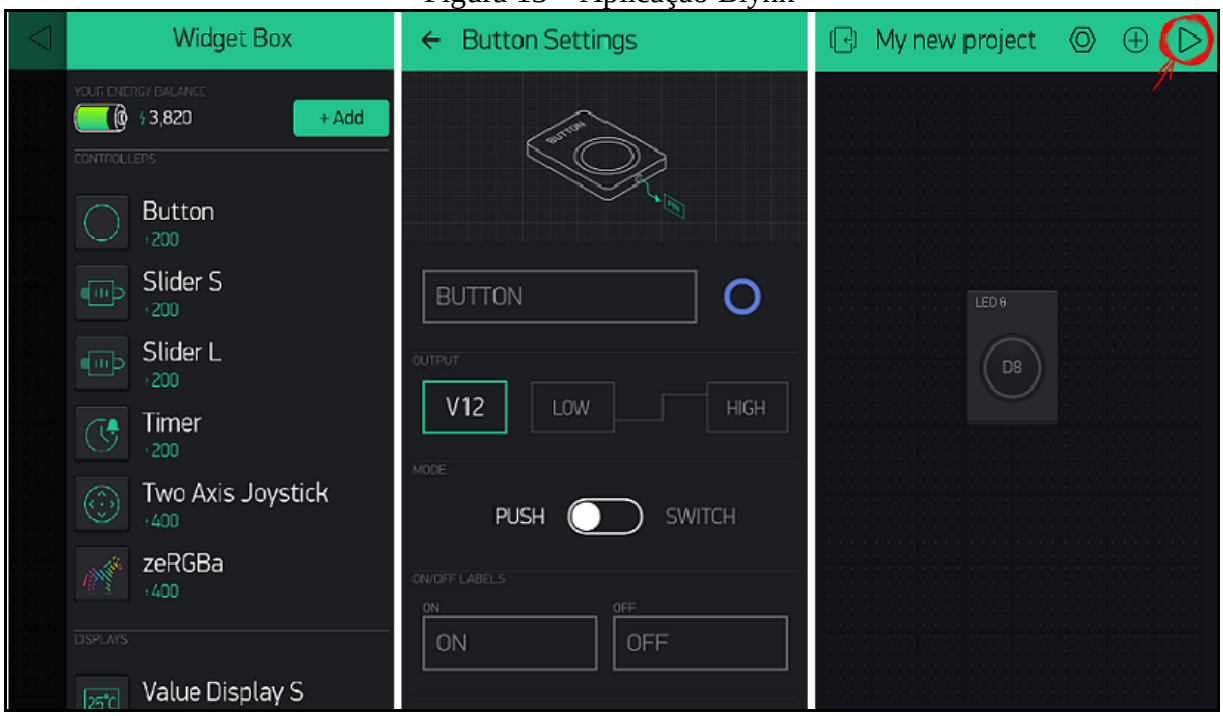
O ambiente do Brainpad possibilita a criação de programas por meio de componentes gráficos, que representam cada ação executada pelo componente eletrônico, ou pela escrita de código nas linguagens de programação JavaScript, Scratch, C# ou Visual BASIC. O *software* possui três níveis de programação, sendo que o nível um (Figura 12), permite a programação através de componentes gráficos, por meio do mecanismo arrasta-e-solta, incorporando comandos idênticos à opção para controlar os sensores, sendo que cada instrução é armazenada para posterior compilação e envio ao componente eletrônico. O nível 2 permite a alteração do código fonte, conforme a construção dos blocos, é gerado linguagem JavaScript ao qual possibilita a intervenção direta ao código. O Nível 3 é relacionado diretamente à compilação e permite que o usuário programe utilizando a linguagem C# ou Visual BASIC e processe diretamente no BrainPad. O terceiro nível permite ainda que a programação seja feita através do Visual Studio.

2.7.2 Blink

Blink (2019) foi desenvolvida pelo Baiborodin para ser uma aplicação para Android que realiza a comunicação com algumas plataformas embarcadas. Inicialmente, o usuário deve emparelhar um aparelho Android com um Arduino para, em seguida, utilizar uma interface para construir a aplicação. Ao ser executada, a aplicação envia as instruções ao

Arduino, que realiza as ações. A Figura 13 apresenta algumas telas da aplicação Blynk, como a tela de componentes, configuração dos componentes e a tela principal da criação do projeto.

Figura 13 – Aplicação Blynk



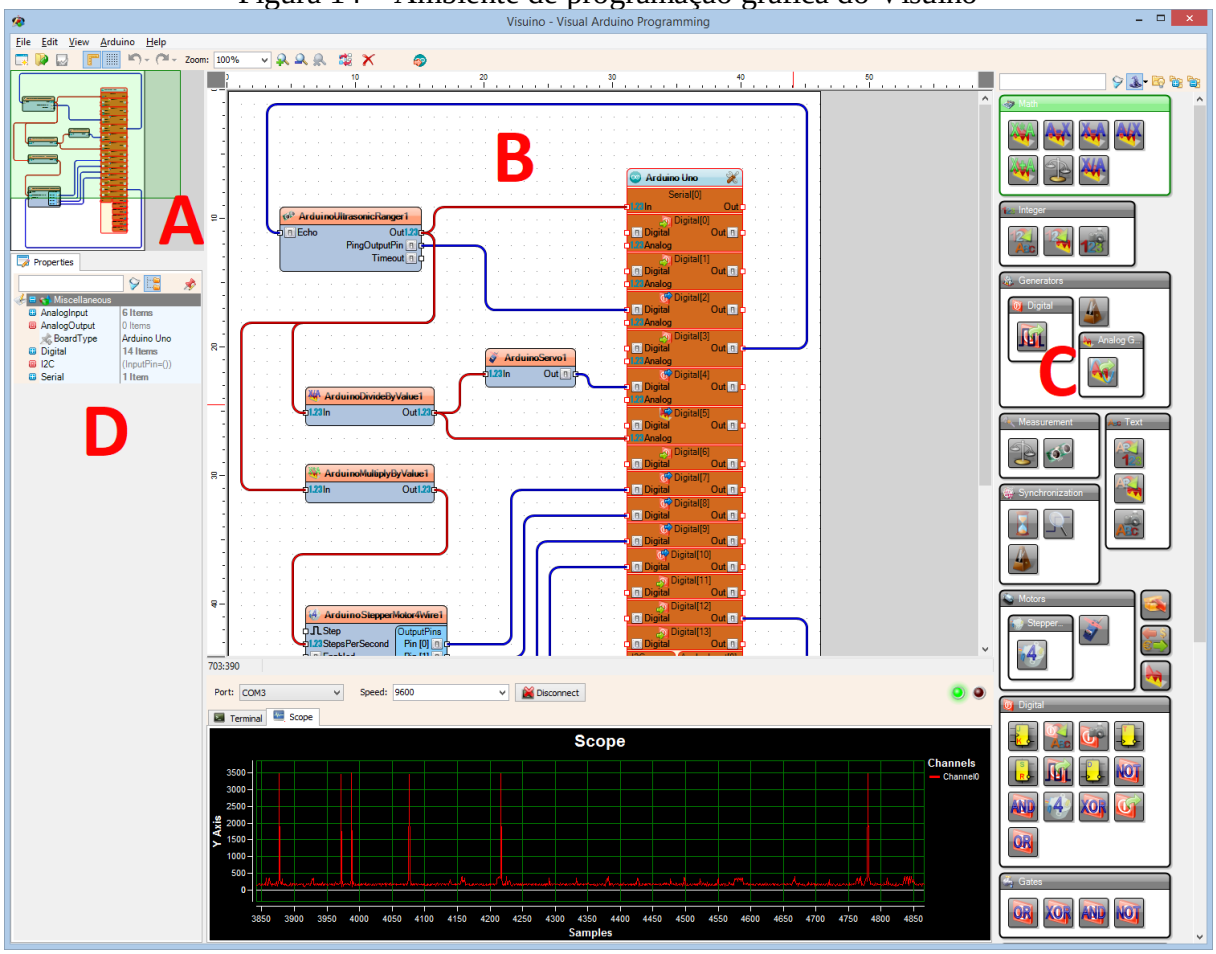
Fonte: Baiborodin (2019).

A aplicação salva todos os seus projetos criados, podendo também carregar projetos já feitos. A tela principal possui uma grade para o encaixe dos blocos sendo que a quantidade de blocos é ilimitada, podendo obter diversos sensores ao mesmo tempo. Além disso, os componentes podem ser ajustados de tamanho. Para a execução do projeto é preciso que o componente eletrônico esteja ligado e pareado para que consiga executar as ações do Blynk. O Blynk, também possui funcionalidades de envio de e-mail, postagem no Twitter e Mapa utilizando o sistema de Global Positioning System (GPS) do dispositivo.

2.7.3 Visuino

O Visuino (VISUINO, 2017) é uma aplicação que utiliza da ferramenta Mitov e foi desenvolvido para ser um ambiente que possibilita a criação de programas para Arduino, utilizando componentes gráficos construídos através de blocos. Na Figura 14 pode-se observar o ambiente do Visuino contendo uma estrutura com alguns componentes. A edição dos componentes é vista do lado esquerdo, abaixo do ambiente de visualização e as funcionalidades são encontradas à direita, contendo diversos blocos.

Figura 14 – Ambiente de programação gráfica do Visuino



Fonte: Visuino (2017).

A aplicação permite que os usuários integrem blocos à estrutura, contendo diversos mecanismos já implementados. Também é possível desenvolver aplicações para que um robô realize diversas ações. Ao ser executado, o *software* Arduino é aberto já com o código disponível para compilação do código e transmissão para o Arduino.

3 DESENVOLVIMENTO

Este capítulo descreve as etapas do desenvolvimento do protótipo. Na seção 3.1 são apresentados os principais requisitos. A seção 3.2 apresenta a especificação. A seção 3.3 descreve de forma detalhada a implementação. Por fim, a seção 3.4 demonstra os resultados dos testes, sugestões e melhorias do protótipo.

3.1 REQUISITOS

A aplicação desenvolvida deve:

a) permitir que o usuário visualize e edite programas escritos em Blockly (Requisito Funcional - RF);
b) permitir que o usuário exclua todos os blocos (RF);
c) permitir que o usuário salve programas escritos em Blockly (RF);
d) permitir que o usuário carregue programas escritos em Blockly (RF);
e) permitir que o usuário gere código na linguagem BASIC (RF);
f) permitir que o usuário visualize o código BASIC gerado (RF);
g) permitir que o usuário altere entre as linguagens Blockly e BASIC (RF);
h) permitir que o usuário execute o código BASIC gerado na placa LPC11u37 (RF);
i) ser desenvolvida para Windows (Requisito não Funcional - RNF);
j) utilizar a linguagem JavaScript, HTML e CSS (RNF);
k) utilizar a plataforma embarcada LPC11u37 (RNF);
l) utilizar ferramenta BASICTools (RNF);
m) utilizar a API Blocky (RNF).

3.2 ESPECIFICAÇÃO

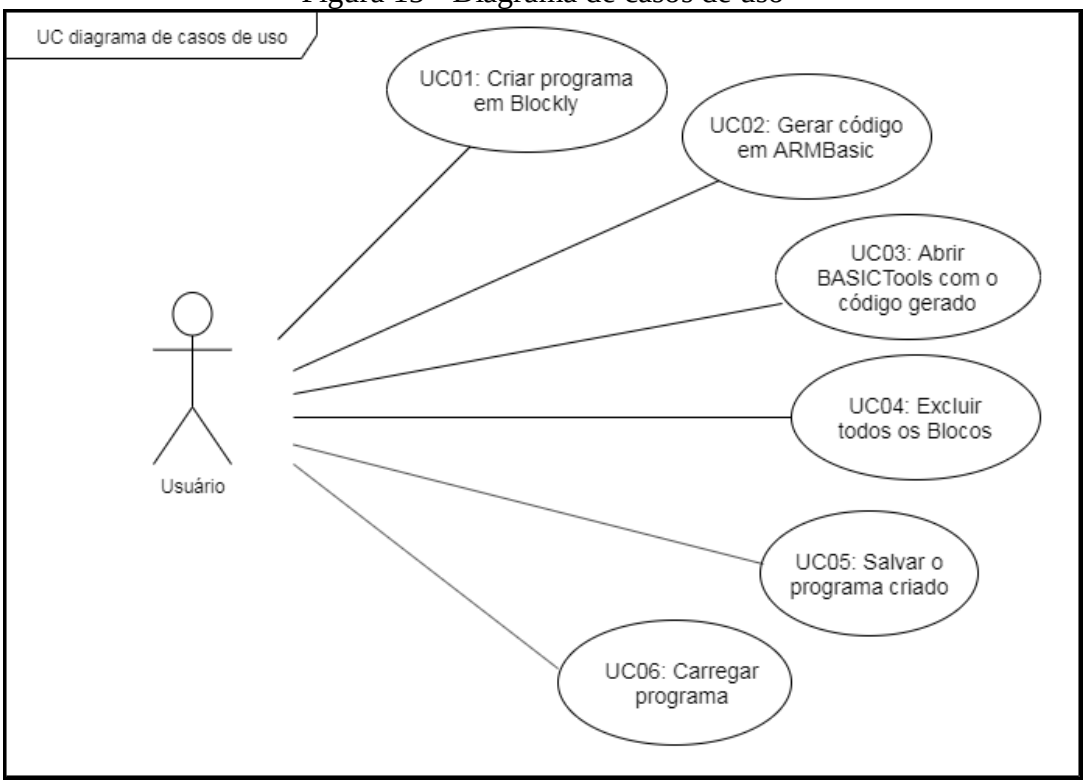
Esta seção apresenta a especificação do protótipo por meio de diagramas da Unified Modeling Language (UML). Utilizou-se o “Draw.io” como ferramenta de desenho. Neste trabalho foram especificados os diagramas de casos de uso e diagrama de sequência.

3.2.1 Casos de uso

Nesta seção são descritos os casos de uso relacionados ao protótipo (Figura 15). Apresenta as principais funcionalidades da aplicação do interfone. Foi identificado apenas um ator, denominado Usuário. No `UC01 – Criar programa em Blockly`, o `Usuário` inicia o protótipo. Neste momento são carregados os blocos, a *Toolbox* e a *Worskpace*. No `UC02 – Gerar código em ARMBasic`, o `Usuário` pode ver o código gerado por um bloco ou

programa construído, para isso é necessário selecionar a aba Basic. No `UC03 – Abrir BASICTools com o código gerado`, o `Usuário` seleciona a ação para executar o código, então o ambiente BASICTools é aberto com o código gerado pelo programa construído no protótipo. No `UC04 – Excluir todos os blocos`, o `Usuário` excluir todos os blocos que estão na *Workspace*, o `Sistema` apresentará um alerta de confirmação. No `UC05 – Salvar o programa criado`, o `Usuário` pode salvar o programa feito no Blockly com a extensão "`.blockly`", para futuramente carregá-lo. Por fim no `UC06 – Carregar programa`, o `Usuário` pode carregar um programa construído no Blockly a partir de um arquivo "`.blockly`".

Figura 15 - Diagrama de casos de uso



Fonte: elaborado pelo autor.

A seguir são descritos os casos de uso. Nas descrições foram omitidas as funcionalidades de cancelar e sair da aplicação.

O caso de uso `Criar programa em Blockly` (Quadro 10), descreve como o usuário pode criar ou editar um programa no protótipo.

Quadro 10 – Caso de uso UC01

UC01 – Criar programa em Blockly: permite ao `Usuário` criar ou editar um programa em Blockly, inserindo, editando ou removendo blocos.	
Requisitos atendidos	RF01.
Pré-condições	Não possui.
Cenário principal	1)O Usuário inicia o programa. 2)O Sistema gera a Workspace, Toolbox e os Blocos 3)Usuário insere, edita ou remove blocos.
Fluxo alternativo 01	Não possui.
Pós-condições	Não possui.

Fonte: Elaborado pelo autor.

O caso de uso `Gerar código em ARMBasic`, apresentado no Quadro 11, descreve como o usuário pode gerar o código em ARMBasic e pode visualizá-lo. O caso de uso possui dois cenários de exceção

Quadro 11 – Caso de uso UC02

UC02 – Gerar código em ARMBasic: permite ao `Usuário` visualize o código gerado pelo sistema.	
Requisitos atendidos	RF03.
Pré-condições	Não possui.
Cenário principal	1)Usuário insere um bloco. 2)O Sistema gera o código em ARMBasic. 3)O Usuário seleciona a aba Basic. 4)O Sistema apresenta o código gerado em forma de texto.
Fluxo de exceção 01	No passo 3 do cenário principal, caso ocorra erro ou alerta na análise semântica. 1) O Software informa o erro ou alerta. 2) Volta ao passo 2 do cenário principal.
Fluxo de exceção 02	No passo 4 do cenário principal, caso ocorra algum erro semântico. 1) Finaliza a geração de código intermediário e apresenta uma tela em branco. 2)Volta ao passo 1 do cenário principal.
Pós-condições	Não possui.

Fonte: Elaborado pelo autor.

O caso de uso `Abrir BASICTools com o código gerado` (Quadro 12) descreve como o usuário pode executar o código gerado pela ferramenta na placa LPC11U37. O caso de uso possui apenas um cenário principal, um cenário alternativo e dois cenários de exceção.

Quadro 12 - Caso de uso UC03

UC03 – Abrir BASICTools com o código gerado: permite ao Usuário executar o código gerado na placa LPC11U37.	
Requisitos atendidos	RF06.
Pré-condições	Não possui.
Cenário principal	1) Usuário seleciona a opção para executar o código. 2) O Sistema gera código em ARMBasic. 3) O Sistema salva o código gerado em um arquivo. 4) O Sistema abre o BASICTools com o código gerado.
Fluxo alternativo 01	No passo 1 do cenário principal, caso o projeto não esteja salvo. 1) Software salvar o projeto em uma pasta dentro do projeto. 2) O Sistema chama um script para abrir o BASICTools .
Fluxo de exceção 01	No passo 1 do cenário principal, caso ocorra erro ou alerta na análise semântica. 1) Software informa o erro ou alerta. 2) Volta ao passo 1 do cenário principal.
Fluxo de exceção 02	No passo 4 do cenário principal, caso a placa não esteja conectada ao computador. 1) Software informa o erro ou alerta. 2) Volta ao passo 1 do cenário principal.
Pós-condições	Não possui.

Fonte: Elaborado pelo autor.

O caso de uso `Excluir todos os blocos` (Quadro 13) descreve como o usuário pode excluir todos os blocos do programa. O caso de uso possui apenas um cenário principal, um cenário alternativo e um cenário de exceção.

Quadro 13 - Caso de uso UC04

UC04 – Excluir todos os blocos: permite ao Usuário excluir todos os blocos.	
Requisitos atendidos	RF03.
Pré-condições	Não possui.
Cenário principal	1) Usuário seleciona a opção para excluir todos os blocos. 2) O Sistema apresenta uma alerta. 3) O Usuário confirma que deseja excluir todos os blocos. 4) O Sistema remove todos os blocos.
Fluxo alternativo 01	No passo 3 do cenário principal, Usuário cancela a operação. 1) Volta para o passo 1 do cenário principal.
Fluxo de exceção 01	No passo 4 do cenário principal, bloco não pode ser removido. 1) O Sistema insere o bloco que não pode ser removido novamente. 2) Volta ao passo 1 do cenário principal.
Pós-condições	Não possui.

Fonte: Elaborado pelo autor.

O caso de uso `Salvar o programa criado` (Quadro 14) descreve como o usuário pode salvar em um arquivo o programa escrito no Blockly. O caso de uso possui apenas um cenário principal, dois cenários alternativos e um cenário de exceção.

Quadro 14 - Caso de uso UC05

UC05 – Salvar o programa criado: permite ao Usuário salvar o programa feito no Blockly.	
Requisitos atendidos	RF03.
Pré-condições	Não possui.
Cenário principal	1) Usuário seleciona a opção para salvar o programa criado no Blockly. 2) O Sistema apresenta uma alerta. 3) O Usuário seleciona o caminho. 4) O Sistema salva o arquivo no caminho selecionado.
Fluxo alternativo 01	No passo 3 do cenário principal, Usuário cancela a operação. 1) Volta para o passo 1 do cenário principal.
Fluxo alternativo 02	No passo 3 do cenário principal, o Usuário salva o arquivo com o mesmo nome de um existente, mas com formato diferente. 1) O Sistema apresenta uma mensagem de erro. 2) Volta para o passo 3 do cenário principal.
Fluxo de exceção 01	No passo 3 do cenário principal, o Usuário salva o arquivo com o mesmo nome de um existente, com o mesmo nome. 1) O Sistema apresenta um alerta. 2) O usuário confirma que deseja substituir o arquivo. 2) Volta para o passo 1 do cenário principal.
Pós-condições	Arquivo salvo.

Fonte: Elaborado pelo autor.

O caso de uso `Carregar programa` (Quadro 15) descreve como o usuário pode carregar um arquivo de um programa escrito no Blockly. O caso de uso possui apenas um cenário principal, um cenário alternativo e um cenário de exceção.

Quadro 15 - Caso de uso UC06

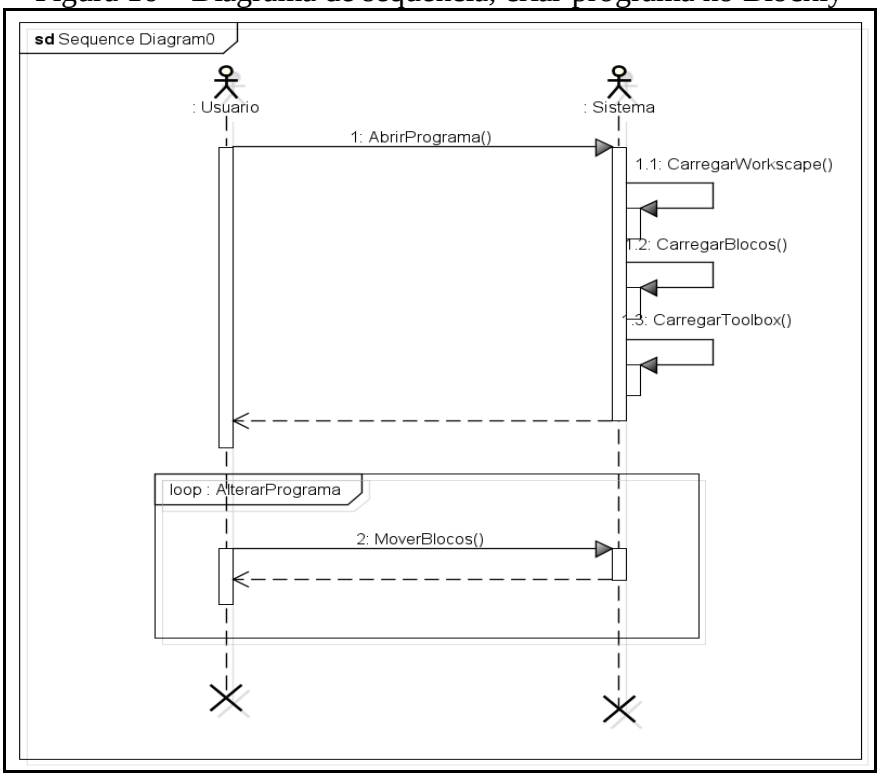
UC06 – Carregar programa: permite ao Usuário carregue um programa feito no Blockly.	
Requisitos atendidos	RF03.
Pré-condições	Programa já salvo.
Cenário principal	1) Usuário seleciona a opção para carregar um programa criado no Blockly. 2) O Sistema apresenta uma alerta. 3) O Usuário seleciona um arquivo. 4) O Sistema carrega o arquivo no caminho selecionado.
Fluxo alternativo 01	No passo 3 do cenário principal, Usuário cancela a operação. 1) Volta para o passo 1 do cenário principal.
Fluxo de exceção 01	No passo 3 do cenário principal, o Usuário carrega o arquivo com formato invalido. 1) O Sistema apresenta uma mensagem de erro. 2) Volta para o passo 2 do cenário principal.
Pós-condições	Programa carregado.

Fonte: Elaborado pelo autor.

3.2.2 Diagrama de sequência

Na Figura 16 apresenta o digrama de sequência onde o usuário abre o programa e o sistema gera Workspace, o Toolbox e os blocos e então retorna para o usuário para que ele possa dar continuidade a criação dos programas.

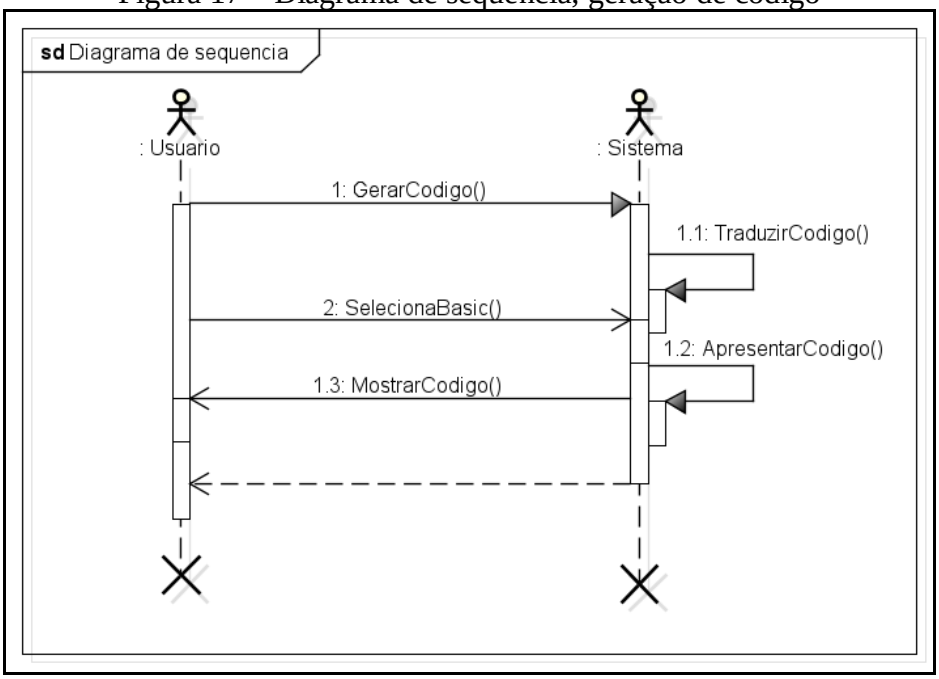
Figura 16 – Diagrama de sequência, criar programa no Blockly



Fonte: Elaborado pelo autor.

A geração de código é apresentada no diagrama de sequência na Figura 17, ao qual o usuário seleciona a aba Basic e o protótipo gera o código correspondente e apresenta na tela em forma de texto.

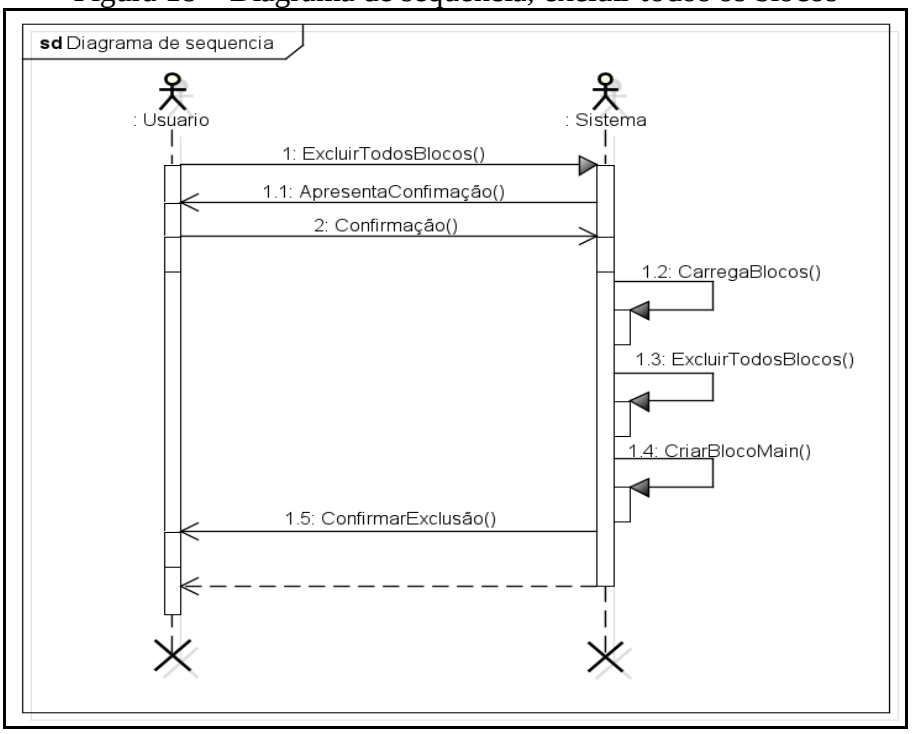
Figura 17 – Diagrama de sequência, geração de código



Fonte: Elaborado pelo autor.

Excluir todos os blocos é apresentado na Figura 18, onde o usuário seleciona a função para exclusão dos blocos e o sistema retorna com uma mensagem de confirmação para que o usuário possa escolher. Caso a resposta seja sim, o sistema carrega todos os blocos existentes na tela e então exclui todos, após isso o sistema cria o bloco `Main:` e o coloca no Workspace. Por ser um bloco necessário para a compilação é preciso sempre o iniciar.

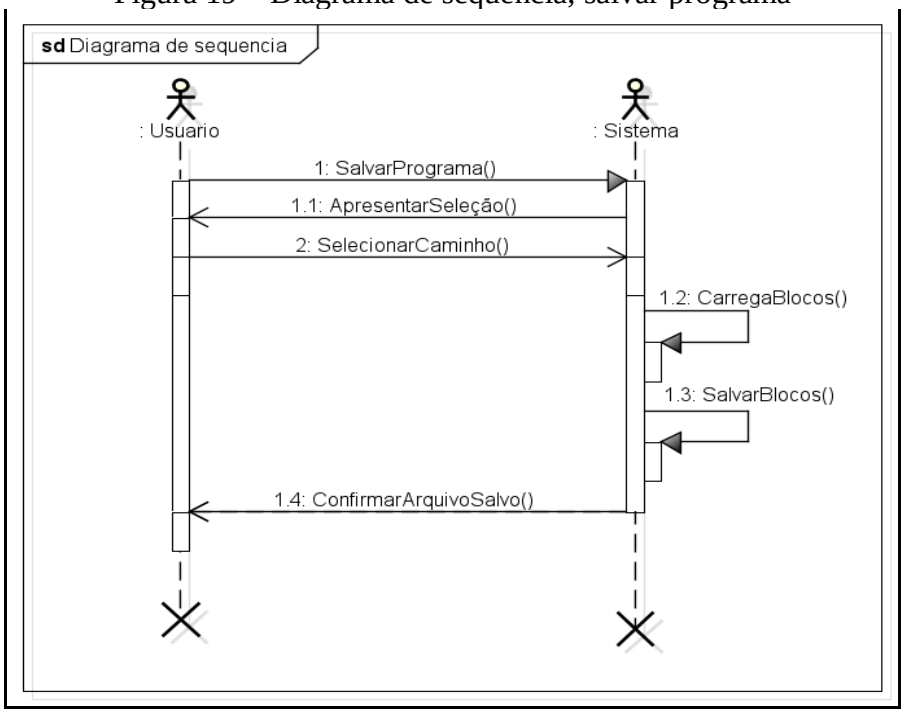
Figura 18 – Diagrama de sequência, excluir todos os blocos



Fonte: Elaborado pelo autor.

No diagrama de sequência da Figura 19, o usuário solicita salvar o programa, pressionando o botão com essa funcionalidade, então o sistema carrega todos os blocos do Workspace, transforma esses blocos em XML e então cria um arquivo, colocando esse bloco no arquivo. O sistema retornará uma mensagem caso o programa seja salvo com sucesso, caso contrário, irá aparecer uma mensagem com o erro correspondente.

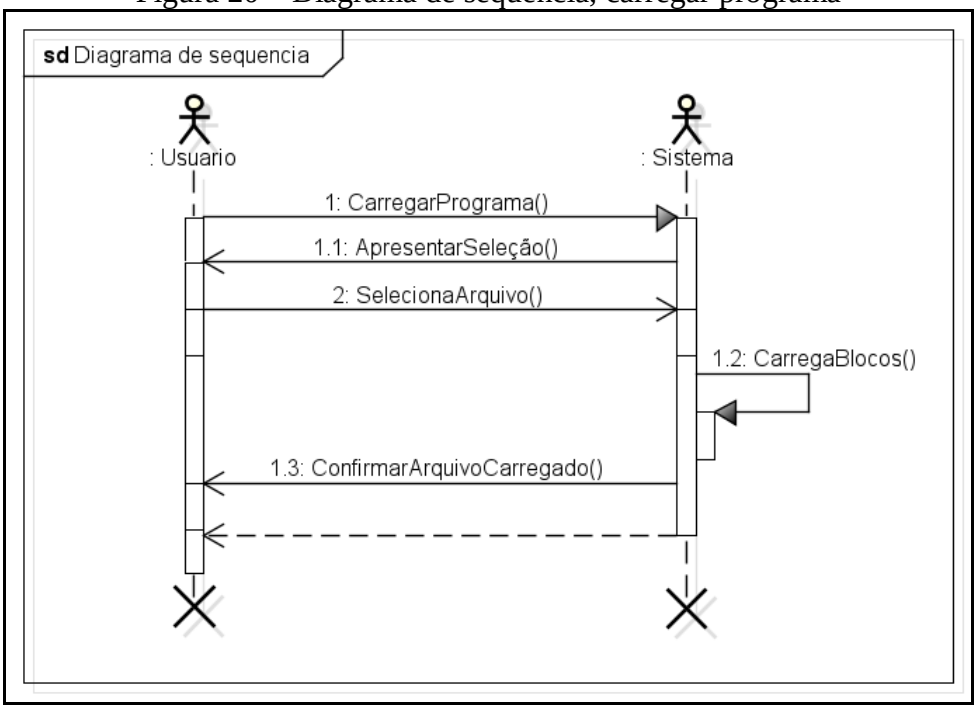
Figura 19 – Diagrama de sequência, salvar programa



Fonte: Elaborado pelo autor.

Por fim, o diagrama de sequência apresentado na Figura 20 apresenta como o usuário interage com o sistema para poder carregar um programa salvo. Solicitando ao sistema para carregar um programa, o usuário confirma através de uma tela retornada pelo sistema, assim o sistema carrega o arquivo, caso o arquivo seja valido e com a extensão apropriada, apresentando assim uma mensagem de sucesso ao carregar o arquivo.

Figura 20 – Diagrama de sequência, carregar programa



Fonte: Elaborado pelo autor.

3.3 IMPLEMENTAÇÃO

Esta seção mostra o detalhamento da implementação do trabalho. A seção 3.3.1 mostra as técnicas e ferramentas utilizadas. A seção 3.3.2 a operacionalidade da implementação, onde é detalho a instalação e a configuração dos *softwares*.

Para melhor entendimento durante a explicação dos módulos, algumas partes do código foram cortadas, para evitar poluição visual de código não relacionado ao contexto da explicação. Como é uma tecnologia pouco utilizada no curso será explicada a implementação de cada parte, pois todas são consideradas importantes.

3.3.1 Técnicas e ferramentas utilizadas

Foram necessárias várias funcionalidades do Blokcly para o desenvolvimento do protótipo. A biblioteca Blockly foi utilizada para gerar código Basic, o Electron foi utilizado para transformar o protótipo em uma aplicação Desktop. O compilador BASICtools

disponibilizado pela Coridium foi utilizado para traduzir o código Basic gerado pelo protótipo em linguagem de máquina para que possa ser interpretado pela placa LPU11U37.

3.3.1.1 Criar blocos e gerar código correspondente

O Quadro 16 apresenta a configuração utilizada para construir a interface gráfica, na *tag* `<head>` conterá os scripts, sendo o blockly_compressed.js, os blocos e os gerados do Basic. No `<body>` foi mantido a aba blocos e adicionado a aba Basic, também foi adicionado os botões `runButton`, `saveButton` e `loadButton`. Por fim, os blocos foram incluídos no `<xml>` onde conterá a categoria dos blocos e os blocos contidos nela.

Quadro 16 – Estrutura HTML da interface gráfica

```
<html>
<head>
      <script src="../../blockly_compressed.js"></script>
      <script src="../../blocks/colour.js"></script>
      <script src="../../generators/basic.js"></script>
</head>
<body>
      <td id="tab_blocks" class="tabon">...</td>
      <td class="tabmin"> </td>
      <td id="tab_basic" class="taboff">Basic</td>
      <td class="tabmin"> </td>
      <xml id="toolbox" style="display: none;">
         <category name="Logic" colour="#5C81A6">
            <block type="controls_if"></block>
            <block type="logic_compare">
                  <field name="OP">EQ</field>
            </block>
            <block type="logic_operation">
                  <field name="OP">AND</field>
            </block>
            <block type="logic_negate"></block>
            <block type="logic_boolean">
                  <field name="BOOL">TRUE</field>
            </block>
         </category>
      </xml>
</body>
</html>
```

Fonte: Elaborado pelo autor.

O Quadro 17 apresenta a relação entre as várias categorias e os blocos que estão incluídos às mesmas, nos próximos parágrafos serão descritos os passos para construí-los.

Quadro 17 – Blocos e categorias atuais

	Categorias							
	Logic	Loops	Math	Text	Variables	Function	Hardware	Vector
Blocos	IF	For	Number	String	Set var	Sub	Wait	DIM Local
	=, !=, <, =<, >, =>	While	Arithmetic	Print text	Get var	Function	AD	DIM Global
	And, OR	For Repeat					Input	Get value in Vector
	Not						Output	Set value in Vector
							IN	
							OUT	
							Pointer	

Fonte: Elaborado pelo autor.

O bloco `IF` está contido na categoria `Logic` e foi construído utilizando o Blockfactory. Na Figura 21 é apresentada a estrutura do bloco `IFELSE` construído as marcações vermelhas sobre os blocos, sinalizam os identificadores (ID) dos blocos e dos respectivos campos. Onde as letras sinalizam:

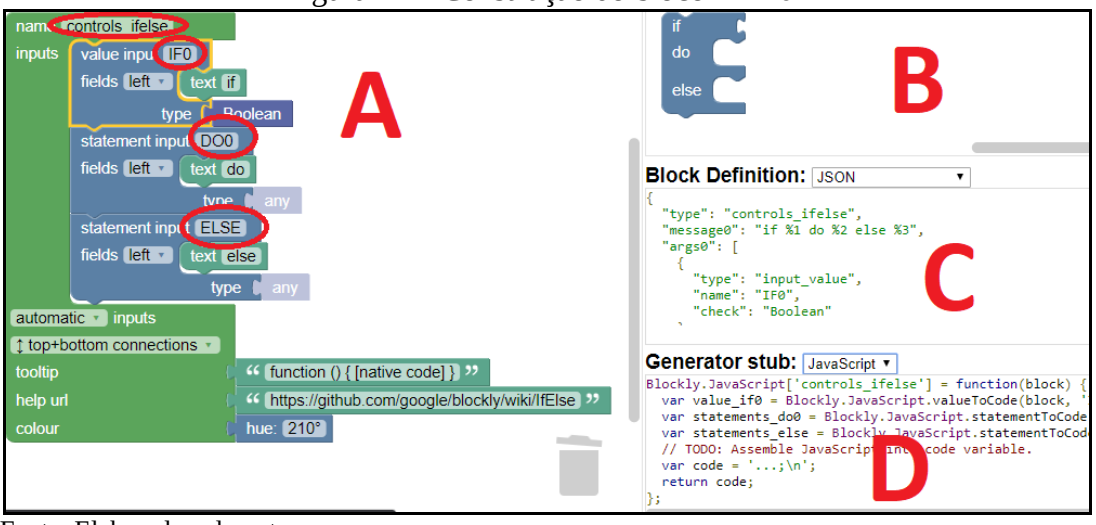
a) Construção do bloco: O bloco possuirá um valor de entrada, sendo restringido a `boolean`, onde seria a condição do `IF`.

b) `Preview`: É apresentada o visual do bloco que está sendo construído.

c) `Block Definition`: Neste quadro é apresentado o código do bloco.

d) `Generator stub`: É apresentado a estrutura do gerador de código, onde será utilizado para definir o código gerado a partir do bloco.

Figura 21 – Construção do bloco `IFELSE`



Fonte: Elaborado pelo autor.

O bloco resultante, após a construí-lo pelo Blockfactory e incluí-lo no “index.html”, está apresentado na Figura 22.

Figura 22 – Bloco IFELSE

Fonte: Elaborado pelo autor.

No Quadro 18 é apresentado o código em JSON referente a criação do bloco gerado pelo Blockfactory, sendo o `type` o nome do bloco, onde neste caso será `controls_ifelse`, o bloco possui três entradas, sendo uma delas a condição e o `input_statement` que seria o corpo das condição do `IF` e do `ELSE`.

Quadro 18 – Código do bloco IFESLE

```
{
    "type": "controls_ifelse",
    "message0": "%{BKY_CONTROLS_IF_MSG_IF} %1",
    "args0": [
      {
        "type": "input_value",
        "name": "IF0",
        "check": "Boolean"
      }
    ],
    "message1": "%{BKY_CONTROLS_IF_MSG_THEN} %1",
    "args1": [
      {
        "type": "input_statement",
        "name": "DO0"
      }
    ],
    "message2": "%{BKY_CONTROLS_IF_MSG_ELSE} %1",
    "args2": [
      {
        "type": "input_statement",
        "name": "ELSE"
      }
    ],
    "previousStatement": null,
    "nextStatement": null,
    "style": "logic_blocks",
    "tooltip": "%{BKYCONTROLS_IF_TOOLTIP_2}",
    "helpUrl": "%{BKY_CONTROLS_IF_HELPURL}",
    "extensions": ["controls_if_tooltip"]
  }
```

Fonte: Elaborado pelo autor.

No arquivo da interface gráfica `index.html`, inclui-se o bloco onde foi estará dentro da categoria `Logic`. Para defini-la utilizou o `type='controls_if'`, que se refere ao nome dado ao bloco na criação usando o Blockfactory.

Quadro 19 – Incluir o bloco na interface gráfica

```
<category name="Logic" colour="#5C81A6">
      <block type="controls_if"></block>
</category>
```

Fonte: Elaborado pelo autor.

O gerador de código Basic correspondente ao bloco `controls_if` está apresentado no Quadro 20, em que o valor da condição do bloco identificador como `IF`, caso não tenha uma condição, será assumido como `false`.

Quadro 20 – Código do gerador de Basic do bloco `controls_if`

```
Blockly.Basic['controls_if'] = function(block) {
  // If/elseif/else condition.
  var n = 0;
  var code = '', branchCode, conditionCode;
  do {
    conditionCode = Blockly.Basic.valueToCode(block, 'IF' + n,Blockly.Basic.ORDER_NONE) || 'false';
    branchCode = Blockly.Basic.statementToCode(block, 'DO' + n);
    code += (n > 0 ? ' ELSE' : '') + 'IF ' + conditionCode + ' THEN\n' + branchCode;
    ++n;
  } while (block.getInput('IF' + n));

  if (block.getInput('ELSE')) {
    branchCode = Blockly.Basic.statementToCode(block, 'ELSE');
    code += ' ELSE\n' + branchCode;
  }
  return code + 'ENDIF\n';
};
```

Fonte: Elaborado pelo autor.

O código gerado pelo bloco `IFELSE` está apresentado no Quadro 21.

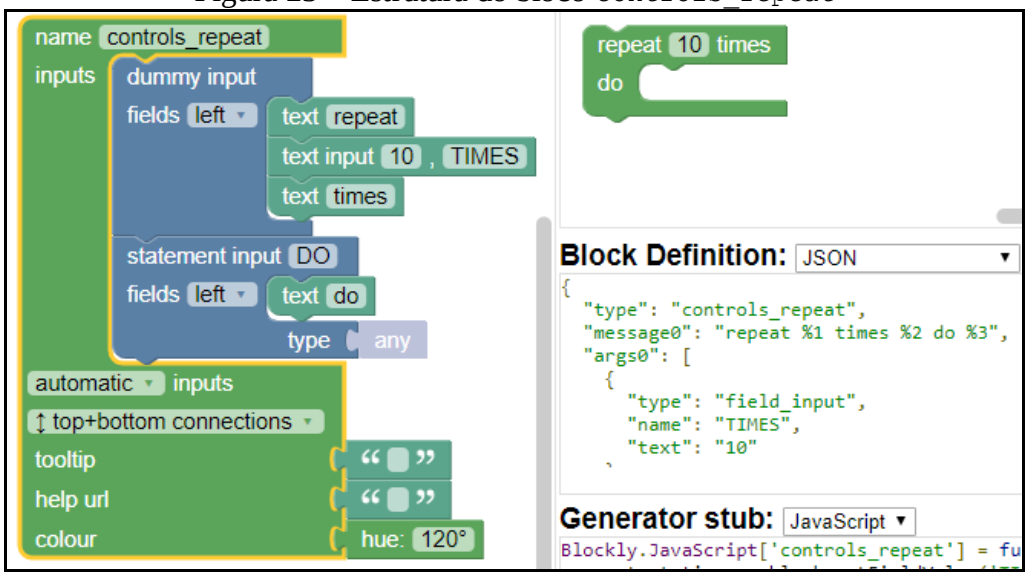
Quadro 21 – Código gerado pelo bloco `IFELSE`

```
IF false THEN
 ELSE
ENDIF
```

Fonte: Elaborado pelo autor.

O bloco `FOR` contém uma estrutura onde uma entrada ao qual é o máximo do incremento e o `DO`, que conterá todo os blocos incluído sobre ele. A estrutura está apresentada na Figura 23.

Figura 23 – Estrutura do bloco `controls_repeat`



Fonte: Elaborado pelo autor.

Como resultado, obteve-se o bloco apresentado na Figura 24.

Figura 24 – Bloco `controls_repeat`

Fonte: Elaborado pelo autor.

O código do bloco que foi gerado pelo Blockfactory está presente no Quadro 22. Como restrição de mínimo foi posto o valor zero, pois sem isso, o protótipo aceitaria valores negativos.

Quadro 22 – Código do bloco `controls_repeat`

```
{
    "type": "controls_repeat",
    "message0": "%{BKY_CONTROLS_REPEAT_TITLE}",
    "args0": [{
      "type": "field_number",
      "name": "TIMES",
      "value": 10,
      "min": 0,
      "precision": 1
    }],
    "message1": "%{BKY_CONTROLS_REPEAT_INPUT_DO} %1",
    "args1": [{
      "type": "input_statement",
      "name": "DO"
    }],
    "previousStatement": null,
    "nextStatement": null,
    "tooltip": "%{BKY_CONTROLS_REPEAT_TOOLTIP}",
    "helpUrl": "%{BKY_CONTROLS_REPEAT_HELPURL}"
  }
```

Fonte: Elaborado pelo autor.

O gerador de código Basic referente ao bloco `controls_repeat` está apresentado no Quadro 20. O gerador também possuí o código do `FOR` onde os valores devem ser colocados manualmente, selecionando qual será o valor do incremento e a variável inicial.

Quadro 23 – Código do gerador de Basic do bloco `controls_repeat`

```
Blockly.Basic['controls_repeat_ext'] = function(block) {
  // Repeat n times.
  if (block.getField('TIMES')) {
    // Internal number.
    var repeats = String(Number(block.getFieldValue('TIMES')));
  } else {
    // External number.
    var repeats = Blockly.Basic.valueToCode(block, 'TIMES',
        Blockly.Basic.ORDER_ASSIGNMENT) || '0';
  }
  var branch = Blockly.Basic.statementToCode(block, 'DO');
  branch = Blockly.Basic.addLoopTrap(branch, block.id);
  var code = '';
  var loopVar = Blockly.Basic.variableDB_.getDistinctName('interaction_loop_count', Blockly.Variables.NAME_TYPE);
  var endVar = repeats;
  if (!repeats.match(/^\w+$/) && !Blockly.isNumber(repeats)) {
    var endVar = Blockly.Basic.variableDB_.getDistinctName('repeat_end', Blockly.Variables.NAME_TYPE);
    code += endVar + ' = ' + repeats + '\n';
  }
  code += 'FOR ' + loopVar + ' = 1 ' + ' TO ' + endVar + ' STEP 1'+
  '\n' + branch;
  return code + 'NEXT ' + loopVar + '\n';
};
```

Fonte: Elaborado pelo autor.

O código resultante do bloco `controls_repeat` está apresentado no

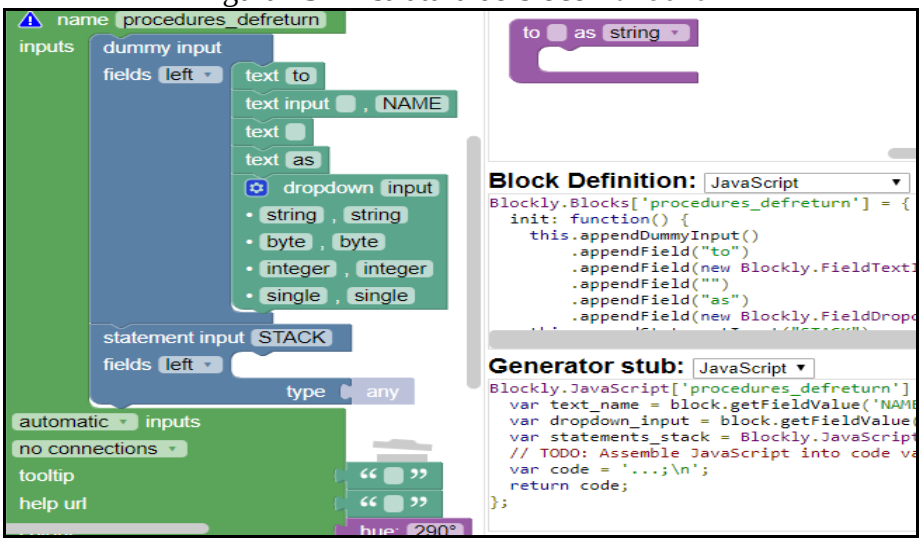
Quadro 24 – Código gerado pelo bloco `controls_repeat`

```
FOR interaction_loop_count = 1  TO 10 STEP 1
NEXT interaction_loop_count
```

Fonte: Elaborado pelo autor.

O bloco `Function`, foi construída utilizando o JavaScript, diferente dos demais. É necessário incluir atributos, para que assim possa ser utilizado no método principal da linguagem `basic.js`. Na Figura 25 é apresentada a estrutura do bloco após sua construção.

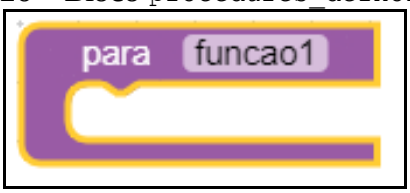
Figura 25 – Estrutura do bloco `Function`



Fonte: Elaborado pelo autor.

O bloco resultante após isso, está apresentado na

Figura 26 – Bloco `procedures_defnoreturn`



Fonte: Elaborado pelo autor.

O código do bloco `Function` referenciada como `procedures_defreturn`, está apresentada no Quadro 25.

Quadro 25 – Código do bloco `procedures_defreturn`

```
Blockly.Blocks['procedures_defreturn'] = {
  init: function() {
    var nameField = new Blockly.FieldTextInput('',
        Blockly.Procedures.rename);
    nameField.setSpellcheck(false);
    this.appendDummyInput()
        .appendField(Blockly.Msg['PROCEDURES_DEFRETURN_TITLE'])
        .appendField(nameField, 'NAME')
        .appendField('', 'PARAMS')
            .appendField("as")
            .appendField(new Blockly.FieldDropdown([["string","string"],
["byte","byte"], ["integer","integer"], ["single","single"]]), "input");
    this.setMutator(new Blockly.Mutator(['procedures_mutatorarg']));
    this.setStyle('procedure_blocks');
    this.setTooltip(Blockly.Msg['PROCEDURES_DEFRETURN_TOOLTIP']);
    this.setHelpUrl(Blockly.Msg['PROCEDURES_DEFRETURN_HELPURL']);
    this.arguments_ = [];
    this.argumentVarModels_ = [];
    this.setStatements_(true);
    this.statementConnection_ = null;
  }
};
```

Fonte: Elaborado pelo autor.

No Quadro 26 é apresentado a estrutura do gerador de código do bloco `procedures_defnoreturn`. Diferente dos demais geradores, por esse bloco ser uma função e não pode estar incluído no bloco dentro do bloco `Main`, o comando `Blockly.Basic.definitions_[]` faz com que o código do bloco `Function` apareça antes do bloco `Main`, mesmo ponto após.

Quadro 26 - Código do gerador de Basic do bloco `procedures_defnoreturn`

```
Blockly.Basic['procedures_defnoreturn'] = function(block) {
  var funcName = Blockly.Basic.variableDB_.getName(
      block.getFieldValue('NAME'), Blockly.Procedures.NAME_TYPE);
  var branch = Blockly.Basic.statementToCode(block, 'STACK');
  if (Blockly.Basic.STATEMENT_PREFIX) {
    var id = block.id.replace(/\$/g, '$$$$');  // Issue 251.
    branch = Blockly.Basic.prefixLines(
        Blockly.Basic.STATEMENT_PREFIX.replace(/%1/g, '\'' + id + '\''), Blockly.Basic.INDENT) + branch;
  }
  if (Blockly.Basic.INFINITE_LOOP_TRAP) {
    branch = Blockly.Basic.INFINITE_LOOP_TRAP.replace(/%1/g, '\'' + block.id + '\'') + branch;
  }
  var code = 'SUB ' + funcName + '\n' + branch;
  code += 'ENDSUB';
  code = Blockly.Basic.scrub_(block, code);
  Blockly.Basic.definitions_['%' + funcName] = code;
  return null;
};
```

Fonte: Elaborado pelo autor.

O código gerado pelo bloco `procedures_defnoreturn` é apresentado no Quadro 27.

Quadro 27 – Código gerado pelo bloco `procedures_defnoreturn`

```
SUB funcao
ENDSUB
```

Fonte: Elaborado pelo autor.

A função `finish`, presente no arquivo `basic.js` como é apresentada no Quadro 28, possui três vetores, onde são armazenadas as precedências de cada código. O primeiro define todas as funções declaradas no código feito pelo Blockly, e os demais contendo todas as instruções que não estão dentro de alguma função. O `defines` é utilizado especificamente para a linguagem Basic, pois há códigos que são executados no pré-processamento, e as funções são inseridas antes de qualquer código.

Quadro 28 – Código fonte, função `finish`

```
Blockly.Basic.finish = function(code) {
  // Convert the definitions dictionary into a list.
  var definitions = [];
  var functions = [];
  var teste = [];
  for (var name in Blockly.Basic.definitions_) {
      var teste = [];
        teste = Blockly.Basic.definitions_[name].split(' ');
    if(teste[0] == "function" || teste[0] == "SUB"){
        functions.push(Blockly.Basic.definitions_[name]);
    } else {
        definitions.push(Blockly.Basic.definitions_[name]);
    }
  }
  var defines = [];
  defines.push("const true = 1","const false = 0","const HIGH = 1","const LOW = 0");
  // Clean up temporary data.
  delete Blockly.Basic.definitions_;
  delete Blockly.Basic.functionNames_;
  Blockly.Basic.variableDB_.reset();
  return defines.join('\n')+'\n'+definitions.join('\n')+'\n'+functions.join('\n')+'\n'+code;
};
```

Fonte: Elaborado pelo autor.

3.3.1.2 Gerando executável com o Electron

A configuração do arquivo `package.json` é apresentada no Quadro 29. Foram inseridas informações como: nome da aplicação, versão, licença que pode ser incluída utilizando um arquivo. Foi definido o `"main"`, como `renderizador` do projeto, ou seja, a tela principal do Electron. Executando o `"electron ."` no console, os executáveis serão gerados para cada tipo de sistema operacional (Windows x64 ou x32).

Quadro 29 - Configuração do `package.json` Electron

```
{
  "name": "App",
  "version": "1.0.0",
  "description": "App usando Blockly",
  "license": "read",
  "main": "main.js",
  "private": true,
  "author": {
    "name": "Will",
    "email": "will@qualquer.com"
  },
  "keywords": [
    "Electron",
    "quick",
    "start",
    "tutorial",
    "demo"
  ],
  "scripts": {
    "start": "electron .",
    "dist": "electron-packager ."
  },
  "engines": {
    "node": ">=8.0.0"
  },
  "dependencies": {
    "debug": "^4.1.1",
    "dialogs": "^2.0.1",
    "electron": "^6.0.7",
    "electron-prompt": "^1.3.1",
    "fs-extra": "^8.1.0",
    "slash": "^3.0.0"
  },
  "devDependencies": {}
}
```

Fonte: Elaborado pelo autor.

Para melhor compreensão de como funciona a estrutura, o arquivo `Main.js`, foi dividido em partes.

No Quadro 30 são apresentadas as declarações que são necessárias ao protótipo.

a) o `app` é a aplicação.

b) `BrowserWindow` é a tela da aplicação, ao qual poderá ter atributos como: tamanho da tela, cor de fundo, tamanho da fonte, entre outros.

c) o `Menu`, adicionara uma barra que conterá funcionalidades que vem com o Electron por padrão caso não seja inserido ou alterado os atributos dele.

d) o `Path` possui o caminho canônico do arquivo `Main.js`, sendo importante para manter o caminho padrão, ignorando a quantidade de pastas anteriores.

Quadro 30 - Arquivo `Main.js`, declarações

```
const { app, BrowserWindow, Menu, dialog } = require('electron')
const fs = require('fs');
var path = process.cwd();
var pathname = path.split("\\").join("/");
```

Fonte: Elaborado pelo autor.

Os atributos principais do `BrowserWindow` são apresentados no Quadro 31, onde:

a) `win`: janela principal do sistema, onde é instanciada pelo `BrowserWindow`;

b) `title` que aparecerá como nome da aplicação na parte superior;

c) `width` e `height` que terá como valor o tamanho inicial da janela;

d) `Modal` recebe como valor `true` fazendo com que a tela possa ser redimensionada;

e) `minWidth` e `minHeight` foram definidos como tamanho mínimo da interface gráfica, onde não pode ser diminuída menos do que foi definido nesses atributos;

f) o `icon` contém o caminho do ícone da aplicação, onde está na mesma pasta do arquivo `main`;

g) `autoHideMenuBar` faz o Menu adicionado na aplicação esconder-se caso seu valor seja `true` e aparecerá somente ao pressionar `alt`;

Quadro 31 - Arquivo Main.js, `BrowserWindow`

```
let win = new BrowserWindow({
      title: 'Coridium Block',
             width: 800,
             height: 600,
      modal: true,
      minWidth: 600,
      minHeight: 400,
      icon: 'core.ico',
      autoHideMenuBar: true.
})
```

Fonte: Elaborado pelo autor.

No Quadro 32, foi definido o `win.loadFile()` para carregar o arquivo da interface gráfica do protótipo.

Quadro 32 - Arquivo Main.js, carregar tela principal e Menu

```
win.loadFile(pathname +
'/resources/app/Blockly/demos/code/index.html')
```

Fonte: Elaborado pelo autor.

Por fim, para habilitar a tela principal do protótipo, foi utilizado o comando `app.on()`, passando como parâmetro um texto que identifique a tela (ID) e a função principal para a criação da interface gráfica (citada anteriormente).

Quadro 33 - Arquivo Main.js, carregar aplicação

```
app.on('ready', createWindow)
```

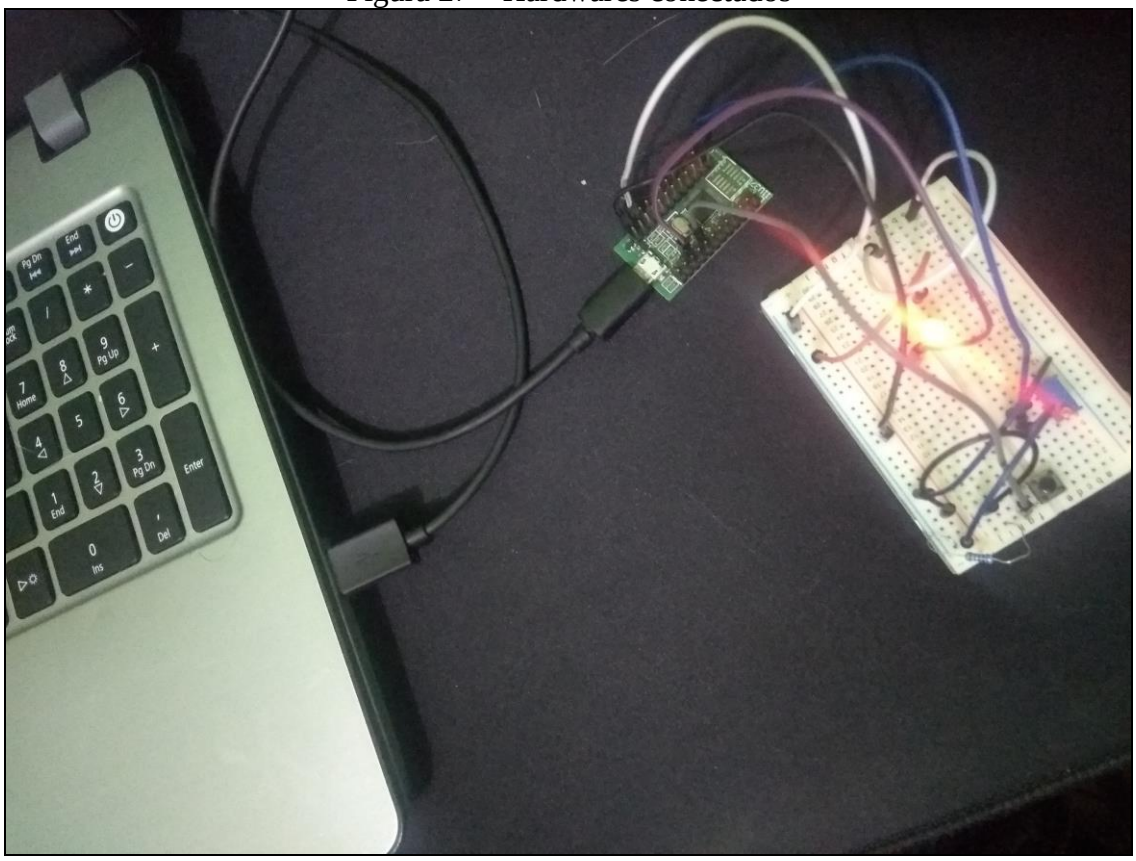
Fonte: Elaborado pelo autor.

3.3.2 Operacionalidade da implementação

Esta seção mostra a operacionalidade da implementação e apresenta um manual simplificado de uso do protótipo desenvolvido.

Inicialmente, deve-se efetuar a conexão da placa com um computador, utilizando um cabo USB. A Figura 27 ilustra o protótipo físico construído para testar as funcionalidades. A placa está conectada ao computador por meio de um cabo USB e o *protoboard* está conectado à placa, na qual contém um LED, um botão e um *trimpot*.

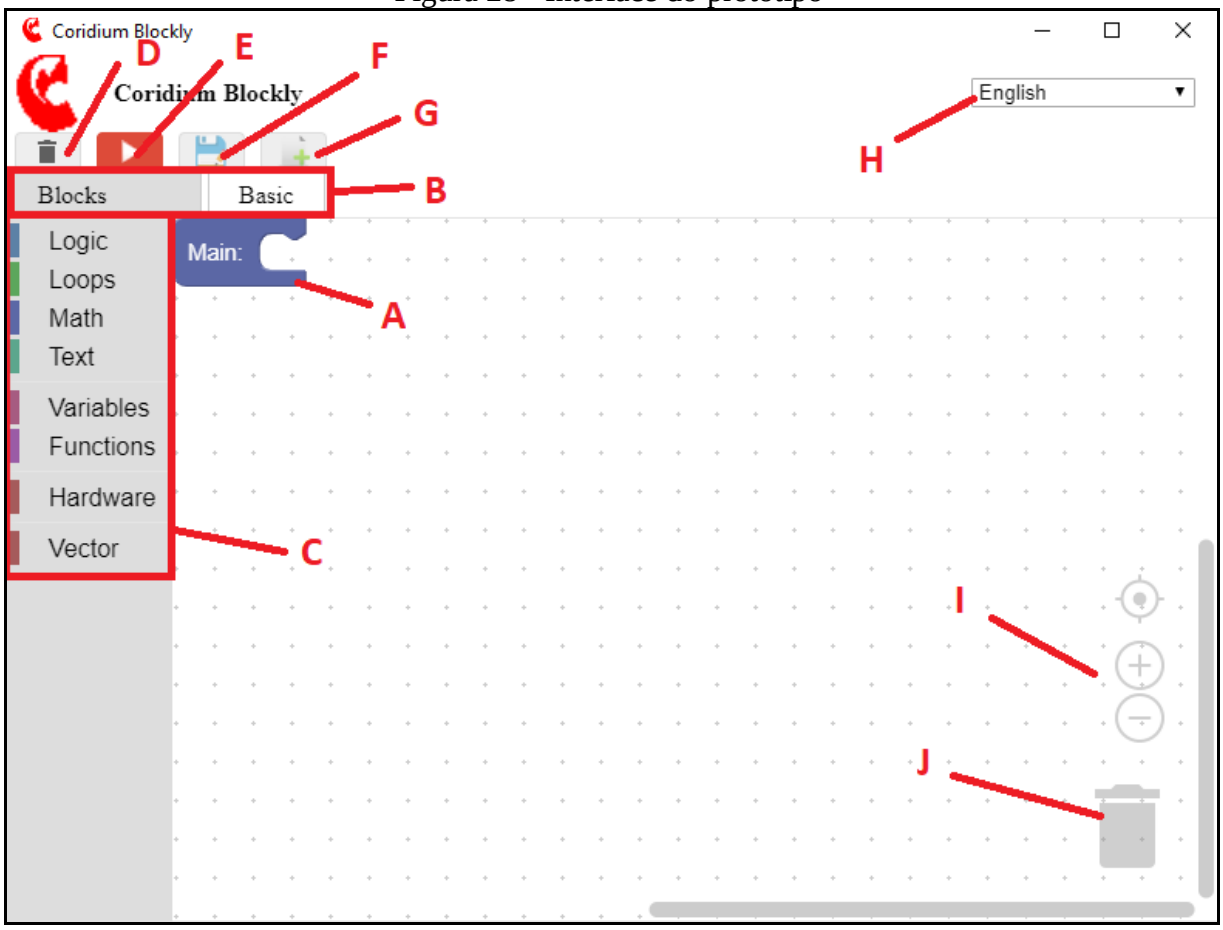
Figura 27 – Hardwares conectados



Fonte: Digitalizado pelo autor.

A interface principal do protótipo é apresentada na Figura 28, removeram-se as linguagens existentes e inseriu-se apenas a linguagem Basic.

Figura 28 - Interface do protótipo



Fonte: Elaborado pelo autor.

É possível identificar os elementos da interface como:

a) Bloco `"MAIN:"` é sempre instanciado como *default* para que o programa sempre possa ser iniciado a partir dele.

b) Abas: Aqui o Usuário poderá alterar entre a interface dos blocos e a interface contendo o código Basic gerado a partir do programa feito pela montagem dos blocos.

c) *Toolbox*: os blocos foram divididos em categoria (*Loop*, *Logic*, entre outros) para facilitar a identificação dos blocos dentro dessas categorias.

d) Excluir todos os blocos: Ao pressioná-lo será exibido uma confirmação perguntando se o Usuário deseja excluir todos os blocos (exceto o *Main*). (caso opte por apagar todos os demais blocos serão excluídos, não sendo possível desfazer a operação).

e) Execução: cria um arquivo ".bas" e abre o BASICTools com o arquivo instanciado. Deve-se observar que a placa precisa estar conectada no computador via USB ou será apresentado um erro ao chamar o BASICTools.

f) Salvar programa: salva o programa em uma pasta selecionada pelo Usuário, a

extensão será “.bas”.

g) Abrir programa: abre um programa por meio da seleção do arquivo pelo Usuário.

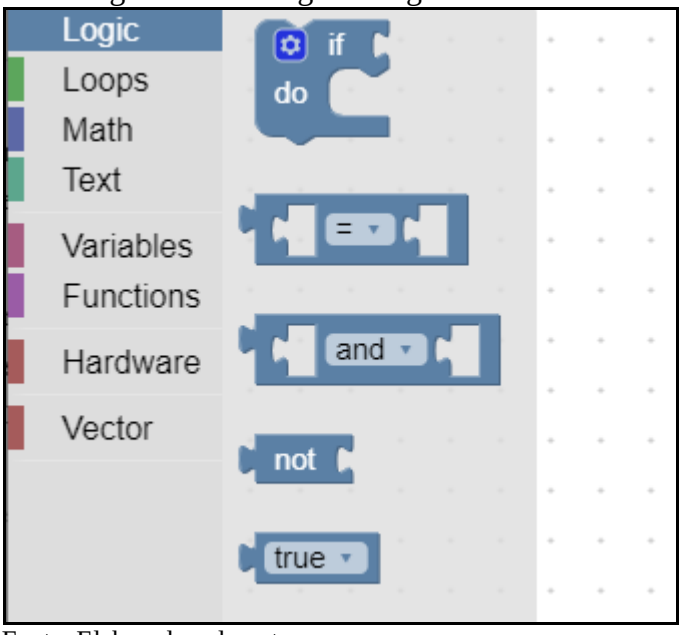
h) Seleção de idioma: a versão final do protótipo só suporta o inglês, mas é possível incluir suporte para outros idiomas.

i) *Zoom*: é possível aumentar ou diminuir o tamanho da área de visualização do protótipo facilitando a operação posicionamento do Usuário no *Workspace*. Para retornar à posição original basta clicar no primeiro botão.

j) Exclusão manual: permite que o bloco atualmente selecionado seja excluído. Esta operação pode ser desfeita.

O primeiro passo para criar um programa no protótipo consiste em o Usuário selecionar o tipo (categoria) da instrução presente no bloco. A *toolbox* contém as instruções organizadas por tipo (categoria), além de uma pré-visualização dos blocos ao ser selecionado uma categoria (Figura 29). O editor permite adicionar as instruções no programa por meio do recurso de arrastar e soltar, para isso basta selecionar uma instrução, arrastá-la até uma posição livre no *Workspace* e soltar a instrução.

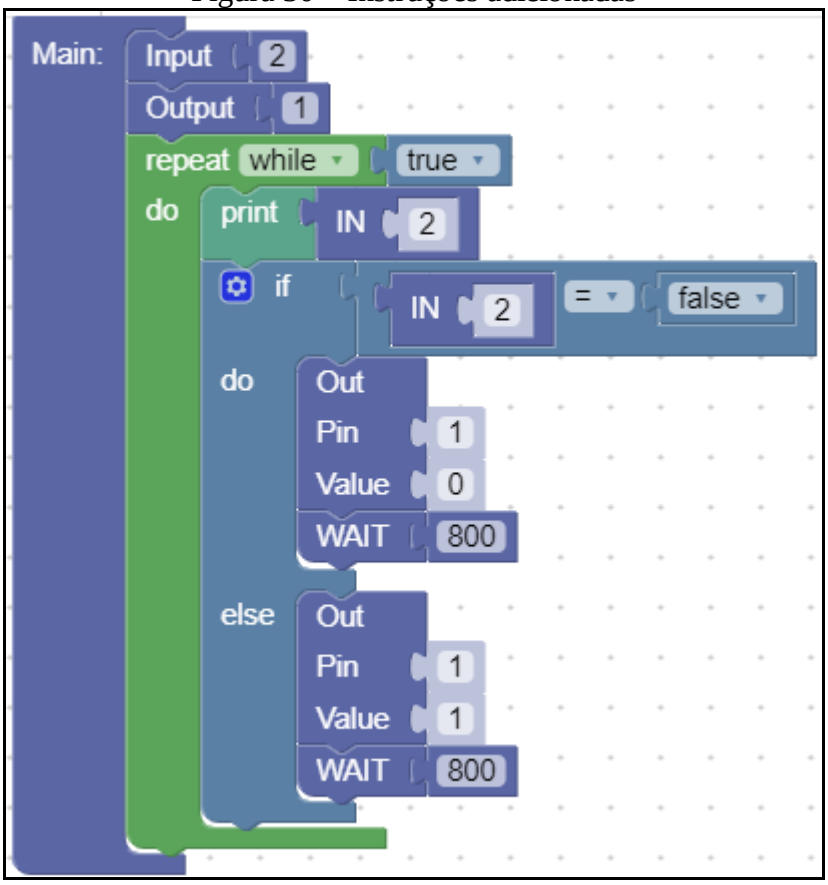
Figura 29 – Categoria Logic na Toolbox



Fonte: Elaborado pelo autor.

Após a inserção das instruções o programa ficará conforme o exemplo ilustrado na Figura 30.

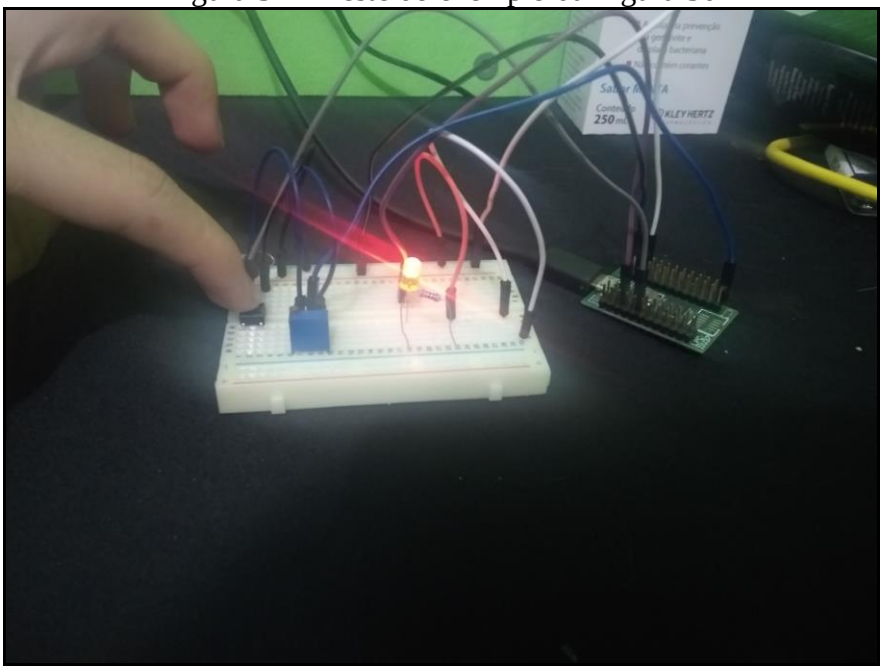
Figura 30 – Instruções adicionadas



Fonte: Elaborado pelo autor

Por fim, o Usuário pode testar a programa por meio da placa de testes (*protoboard*). A execução do exemplo está apresentada na Figura 30; o Usuário manterá pressionado o botão e o LED acenderá como pode ser visto na Figura 31.

Figura 31 – Teste do exemplo da Figura 30



Fonte: Digitalizado pelo autor.

3.4 ANÁLISE DOS RESULTADOS

O escopo inicial do trabalho não restringia qualquer aspecto quanto à quantidade de instruções, porém, foi identificada como um complicador em relação ao fator tempo de desenvolvimento. Como forma de reduzir a complexidade do editor gráfico em relação à quantidade de instruções foi limitado este recurso a apenas instruções formadas por blocos que permitem utilizar-se de componentes básicos disponibilizados pelo *hardware*. Os investimentos necessários para aquisição da placa foram aproximadamente de R$ 100,00 ($25).

Inicialmente a fase de teste era composta apenas pelo LED encontrado na placa, pois o intuito era apenas testar a interface. A partir do sucesso obtido montou-se um protótipo em um *Protoboard* com um LED, leitor de temperatura e um botão, permitindo a criação de programas mais complexos e uma melhor análise em relação ao desempenho do programa Blocklyy.

As seções a seguir relatam o teste de desempenho do programa, o teste de funcionalidade do projeto e o comparativo com os trabalhos correlatos. Na seção 2.5.1 apresenta o teste de usabilidade.

3.4.1 Teste de usabilidade do protótipo

A fim de efetuar uma avaliação das funcionalidades do protótipo, foi elaborado algumas perguntas (Quadro 34) as quais foram encaminhadas diretamente ao dono da empresa Coridium, utilizando como base a interface e a criação de programas usando blocos.

Quadro 34 – Perguntas feitas à Coridium

Perguntas
De onde surgio a ideia de criar um programa visual?
O que achou da versão de avaliação do programa?
O que achou do Visual? Agradável?
Como foi a construção de programas usando blocos? Fácil ou difícil?

Fonte: Elaborado pelo autor.

No que diz respeito à adequação funcional foram elaboradas três afirmações com base nas respostas passados pela empresa:

a) completude funcional: o editor permite criar um programa de acordo com as características da linguagem ARMBasic;
b) corretude funcional: a execução do código gerado corresponde ao programa ARMBasic criado;

c) funcionalidade apropriada: a utilização do editor facilita a criação de código da linguagem ARMBasic;
d) facilidade de uso: o editor gráfico visual possui elementos que facilita a criação de programas, sendo eles complexos ou não.

Todos esses pontos fortes apresentados foram confirmados[6] e parabenizados pela Coridium, ao qual contribui com os testes do programa durante o último mês do projeto, também a partir dessas afirmações conclui-se que a ferramenta criada foi aceita pela empresa e continuará a ser desenvolvida e melhorada.

3.4.2 Comparativo com trabalhos correlatos

O Quadro 35 apresenta de forma comparativa as características dos trabalhos correlatos. Observa-se que todos os trabalhos têm uma interface gráfica e geram código para dispositivos eletrônicos. O Brainpad possui, junto à interface, um ambiente para simulação do programa gerado sem a necessidade do uso de um componente eletrônico. Por outro lado, Blynk e Visuino não possuem esse ambiente, sendo necessário a transferência do programa para um *hardware*.

Quadro 35 – Comparativo entre trabalhos relacionados

correlatos / características	Silveira (2019)	Brainpad (2019)	Blynk (2019)	Visuino (2017)
gera código para mais de um dispositivo	X		X	X
tipo de programação	Blockly	Scratch	Gráfica	Gráfica
placas suportadas	Todas com suporte a ARMBasic	Brainpad	Arduino (e Ramificações), ESP8266, Raspberry PI entre outros.	Arduino (e Ramificações), Raspberry PI, ESP8266, entre outros.
plataformas	Web,Android e Desktop	Web, Android e IOs	Android e iOS	Desktop
intuito educacional	X	X	X	
dependente de hardware	X		X	X
linguagem gerada	ARMBasic, JavaScript, Python, PHP, Lua e Dart	Javascript	C++	C++
idiomas	Suporta mais de 40 idiomas	Inglês	Inglês	Inglês

Fonte: Elaborado pelo autor.

[6] Seu site está presente as notícias sobre o protótipo https://www.coridium.us/coridium/blog/basic-for-microbit

Exceto pelo Brainpad, os demais possuem suporte para diversas plataformas embarcadas, sendo o projeto desenvolvido com base em uma linguagem utilizada por diversos *hardwares*, sendo possível a utilização em diversas placas que suportem o ARMBasic ou o Basic; o Arduino também pode ser incluído, pois possuí também um interpretador Basic.

Um ponto forte com relação aos outros trabalhos propostos é que o Coridium Blockly pode ser rodado em qualquer plataforma, bastando apenas pequenos ajustes em código fonte, por ser WEB. Utilizando APIs para portabilidade, transforma a aplicação para qualquer plataforma, mas para diminuir dificuldades futuras e não consumir muito tempo para desenvolvimento, optou-se em focar no lado Desktop.

Em relação ao item de linguagem gerada, o Coridium Blockly possui seis linguagens em sua geração de códigos, já as demais possuem apenas uma. Como as demais linguagens do Coridium Blockly foram irrelevantes, não foram incluídas, sendo apenas o ARMBasic atualmente, mas com poucas modificações é possível inclui-las novamente. Sendo expansivo para novas linguagens, o Blockly possui uma estrutura capaz de suportar a inclusão de diversas linguagens de programações em sua geração de código.

Por fim, Silveira (2019) o suporte a mais de 40 idiomas, como sendo um dos pontos mais importantes em relação aos trabalhos correlatos apresentados, leva-se em conta as regras relacionadas a cada idioma, sendo uma das regras o alinhamento à esquerda ou a direita.

4 CONCLUSÕES

O presente trabalho teve como objetivo principal disponibilizar uma ferramenta que permita um *hardware* com ARMBasic Embarcado executar códigos enviados por meio de um compilador. Para tal foi desenvolvido um editor gráfico por meio da API Blockly. O editor permite traduzir uma representação intermediária gerada a partir do programa de blocos para código em linguagem ARMBasic, suportado pelo *hardware*. O desenvolvimento do editor gráfico demonstrou ser um desafio em relação à estrutura de arquivos, devido à complexidade para interpretar os dados que são dispostos entre os arquivos JavaScript.

O trabalho atingiu os objetivos de gerar código coerente, carregar um programa na memória do *hardware*, processar as entradas e produzir saídas conforme o esperado. Também foi criada uma documentação detalhada da criação de uma nova linguagem e de novos blocos para que assim fosse possível extensões. Não há grande complexidade em adicionar novas instruções, geradores de linguagens e novos blocos. O Blockly, mesmo que sendo uma API livre, atendeu muito bem ao desenvolvimento da ferramenta, fornecendo todas as funções necessárias. Os elogios da empresa Coridium quanto à interface deixaram claros que um ambiente gráfico com recurso de arrastar e soltar facilita a criação da lógica.

Este trabalho foi relevante por mostrar a possibilidade de implementar uma ferramenta visual com tecnologias livres e *hardwares* de fácil acesso. Também contribuiu para mostrar a possibilidade desse tipo de sistema ser desenvolvido para ser utilizado em minicomputadores de propósito geral. Os testes apresentaram resultados satisfatórios e pode-se concluir que os objetivos propostos foram alcançados.

4.1 EXTENSÕES

Para melhorar as funcionalidades deste trabalho ou acrescentar novas, as seguintes extensões são sugeridas:

e) adicionar novas linguagens de programação na geração de código;
f) desenvolver uma versão para *smartphones* do Blockly, permitindo assim que a aplicação seja roda em dispositivos móveis e através da rede;
g) implementar mais blocos relacionados à linguagem;
h) utilizar um *Web service* para que o Blockly possa ser manipulado através de um navegador;
i) uso de *softwares* "*open-source"* para melhorar a interface, garantir segurança e integridade dos arquivos.

REFERÊNCIAS

ALEXANDRINI, Fábio et al. Desenvolvimento do Compilador da Linguagem Basico. In: SIMPÓSIO DE EXCELÊNCIA EM GESTÃO E TECNOLOGIA, 21., 2014, Rio do Sul. **Anais** [...]. Rio do Sul: IFC, [2014?]. p. 1-13. Disponível em: https://www.aedb.br/seget/arquivos/artigos14/11720295.pdf. Acesso em: 27 nov. 2019.

ARAÚJO, ALAN KILSON RIBEIRO**. INTERNET DAS COISAS: ANÁLISE DAS QUESTÕES CHAVE DE APLICAÇÃO PÚBLICA NO BRASIL**. 2017. Dissertação (Mestrado em Engenharia de Produção) – Instituto de Engenharia de Produção, Universidade Paulista, São Paulo. Disponivel em: < https://www.unip.br/presencial/ensino/pos_graduacao/strictosensu/eng_producao/download/eng_alankilsonribeiroaraujo.pdf>. Acessado em: 03 nov. 2019.

ARDUINO. **What is Arduino?** 2018. Ivrea. Disponível em: <https://www.arduino.cc/>. Acesso em: 24 out. 2019.

BAPTISTA, João Alvaro de Souza. **Programação com Scratch: Desenvolvendo Raciocínio Algorítmico**. 2017. 70 f. TCC (Graduação) - Curso de Matemática em Rede Nacional, Instituro Nacional de Matematica Pura e Aplicada, Rio de Janeiro, 2017. Cap. 6.

BEZERRA, F.; DIAS, K. (2014). **Programação de Computadores no Ensino Fundamental**: Experiências com Logo e Scratch em escola pública. In XXII Workshop sobre Educação em Informática, Brasília, DF: SBC.

BLYNK. **The most popular mobile app for the IOT. Works with anything: ESP8266, Arduino, Raspberry Pi, SparkFun and others.:** 2018. [S.I.]. Disponível em: <https://www.blynk.cc/>. Acesso em: 24 out. 2019.

BRANCO, Guido Aparecido Junior; TAMAE, Rodrigo Yoshio. **UMA BREVE INTRODUÇÃO AO ESTUDO E IMPLEMENTAÇÃO DE COMPILADORES**. Revista científica eletrônica de Psicologia, Garça - SP, v. 10, n. 8, p.1-6, fev. 2008. Semestral. Disponível em: <http://faef.revista.inf.br/imagens_arquivos/arquivos_destaque/RHXqIjJHvJQhhCK_2013-5-28-11-13-48.pdf>. Acesso em: 09 nov. 2019.

CHAVIER, Luís Fernando. **Programação para Arduino - Primeiros Passos**. Disponível em: <https://www.circuitar.com.br/tutoriais/programacao-para-arduino-primeiros-passos/#programa-de-computador>. Acesso em: 07 nov. 2019.

COLÉGIO WEB. **3 meios de comunicação mais famosos do Mundo**. 2013. Disponível em: <https://www.colegioweb.com.br/curiosidades/3-meioscomunicacao-famosos- mundo.html>. Acesso em: 27 nov. 2019.

CORIDIUM. **Leverage our Expertise.** 2018. Tahoe Vista. Disponível em: <https://www.coridium.us/coridium/>. Acesso em: 01 dez. 2019.

DELAMARO, M.E. **Com construir compilador utilizando ferramentas Java**. Ed. Novatec, São Paulo. 2004

DINIZ, Sirley Nogueira de Faria. **O uso das novas tecnologias em sala de aula**. 2001. 186 f. Dissertação (Mestrado) - Curso de Engenharia de Produção, Engenharia de Produção, Universidade Federal de Santa Catarina, Florianópolis - Sc, 2001. Disponível em: <http://www.pucrs.br/ciencias/viali/doutorado/ptic/aulas/aula_2/187071.pdf>. Acesso em: 26 nov. 2019.

DORNELAS, E.; CAMPELLO, S. Monitoramento de consumo doméstico de água utilizando uma meta-plataforma de IoT. **Revista de Engenharia e Pesquisa Aplicada**, v. 2, n. 2, 27 jul. 2017.

ELECTRONICS, Ghi. BrainPad. 2018. Madison Heights. Disponível em: <https://www.ghielectronics.com/products/BrainPad>. Acesso em: 17 nov. 2019.

EVAN, D. **A Internet das Coisas: Como a próxima evolução da Internet está mudando tudo. White paper Cisco**, abril de 2011. Disponível em: < https://www.cisco.com/c/dam/global/pt_br/assets/executives/pdf/internet_of_things_iot_ibsg_0411final.pdf >. Acesso em: 07 nov. 2019.

DEVELOPERS, Google. **A JavaScript library for building visual programming editors.** [2019]. Disponível em: <https://developers.google.com/blockly>. Acesso em: 17 dez. 2019.

GODINHO, et. al., 2017. Disponível em: <https://sol.sbc.org.br/index.php/wei/article/view/3553/3512>. Acesso em 07 nov. 2019.

GOOGLE DEVELOPERS, 2019. Disponível em: <https://developers.google.com/blockly>. Acesso em: 07 nov. 2019.

GOMES, Anabela de Jesus. **Dificuldade de aprendizagem de programação de computadores**: contributos para a sua compreensão e resolução. 2010. 492 f. Dissertação (Mestrado) - Curso de Ciência e Tecnologia, Engenharia Informática, Universidade de Coimbra, Coimbra, 2010. Cap. 9.

GONÇALVES, Débora. **Metodologias ágeis de desenvolvimento de software**: saiba mais sobre o assunto. 2019. Cronapp. Disponível em: <https://blog.cronapp.io/metodologias-ageis-de-desenvolvimento-de-software/>. Acesso em: 29 nov. 2019.

HENTGES, Vanice; BULEGON, Ana Marli. **Programação de Softwares no ensino fundamental e suas contribuições no desenvolvimento do pensamento lógico-matemático: uso do Scratch**: Como ambientes educativos formais e informais formam cidadãos capazes de compreender e cumprir seus deveres pessoais e sociais. 2016. [S.I.]. Disponível em: <https://reciprocidade.emnuvens.com.br/novapedagogia/article/view/177/199>. Acesso em: 25 out. 2019.

LACERDA, F.; LIMA-MARQUES, M**.** Da necessidade de princípios de arquitetura da Informação para a Internet das Coisas. **Perspectivas em Ciência da Informação**, v.20, n.2, p.158-171, abr./jun. 2015

MORIMOTO, Carlos. **BASIC**. 2005. Disponível em: <https://www.hardware.com.br/termos/basic>. Acesso em: 15 nov. 2019.

MICROSOFT. **Hands on computing education.** 2018. Redmond. Disponível em: <https://www.microsoft.com/en-us/makecode>. Acesso em: 24 nov. 2019.

MITOV. **Rapid Development in Audio, Video, DSP & Computer Vision now supporting iOS and Android.** 2013. Moorpark. Disponível em: <http://mitov.com/>. Acesso em: 29 out. 2019.

MOTA, Rafael Perazzo Barbosa; BATISTA, Daniel Macedo. Um mecanismo para garantia de QoS na Internet das Coisas com RFID. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS, 31., 2013, São Paulo. **Anais** [...]. São Paulo: USP, [2013?]. p. 297-310. Disponível em: http://ccsl.ime.usp.br/files/publications/files/2013/sbrc_artigo2013.pdf. Acesso em: 10 dez. 2019.

NUNES, D. J. **Ciência da Computação na Educação Básica**. 2011. Porto Alegre. Disponível em: <http://www.adufrgs.org.br/artigos/ciencia-da-computacao-na-educacao-basica/>. Acessado em: 06 nov. 2019.

PACKARD, Hewlett. **O que é Internet das Coisas?** 2018. Disponível em: <https://www.hpe.com/br/pt/what-is/internet-of-things.html>. Acesso em: 21 nov. 2019.

PEREIRA, J. C. R., RAPKIEWICZ, C. (2004). **O Processo de Ensino-Aprendizagem de Fundamentos de Programação**: Uma Visão Crítica da Pesquisa no Brasil, WEI RJES.

Peterson, L. L., Davie, B. S. (2011). Computer Networks, Fifth Edition: A Systems Approach. **Morgan Kaufmann Publishers** Inc., San Francisco, CA, USA, 5th edition.

SOUSA, A., SILVA, S., RAIOL, A.A.C., SARGES, J.,BEZERRA, F. (2015) **O Universo Lúdico da Programação com Logo no Ensino Fundamental**. In: 23º WEI – Workshop sobre Educação em Computação – CSBC 2015.

SCRATCH. **Crie estórias, jogos e animações**. 2018. [S.l.]. Disponível em: <https://scratch.mit.edu/>. Acesso em: 24 set. 2019.

SILVA, André Luis Zary Mariano da. **Linguagens de Programação - Basic**. 2009. Disponível em: <http://www.vidageek.net/2009/03/16/linguagens-de-programacao-basic/>. Acesso em: 07 nov. 2019.

SOSA, Henrique. **Introdução ao Electron**. 2015. Tableless. Disponível em: <https://tableless.com.br/introducao-ao-electron/>. Acesso em: 23 nov. 2019.

TEIXEIRA, Fernando et al. Siot – Defendendo a Internet das Coisas contra Exploits. *In*: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS, 32., 2014, São Paulo. **Anais** [...]. Belo Horizonte: UFMG, [2014?]. p. 589-602. Disponível em: http://sbrc2014.ufsc.br/anais/files/trilha/ST14-1.pdf. Acesso em: 12 nov. 2019.

TORCZON, Linda; KEITH, Cooper. **Construindo Compiladores**. 2ª. ed. Rio de Janeiro: Elsevier, 2014.

TUCKER, Allen B; NOONAN, Robert E. **Linguagens de programação: princípios e paradigmas**. 2. ed. São Paulo: McGraw-Hill, 2008. 599 p.

VISUINO. **What is Visuino?** 2017. [S.l.]. Disponível em: <http://www.visuino.com/>. Acesso em: 24 set. 2019.

ZABADAL, Bernardo Moreira; LISBOA MURTA DE CASTRO, Bianca Francinny. IoT e Seus Principais Desafios. **Revista Interdisciplinar de Tecnologias e Educação**, [S.l.], v. 3, n. 1, july 2017. ISSN 2447-5955. Disponível em: <http://rinte.ifsp.edu.br/index.php/RInTE/article/view/333>. Acesso em: 17 dez. 2019.

ZAMPIERI, Gabriel. **O que é JavaScript**. 2019. Hostinger. Disponível em: <https://www.hostinger.com.br/tutoriais/o-que-e-javascript/>. Acesso em: 30 nov. 2019.