UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

ESCOLA DE ENGENHARIA

TRABALHO DE CONCLUSÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Infraestrutura para implementação em linguagem C de controladores supervisórios para plantas de manufatura virtuais.

Autor: Felipe Alves Pereira

Orientador: Prof. Dr. Marcelo Götz

Porto Alegre, novembro de 2014

Sumário

Sumário ii
Agradecimentos iv
Resumo v
Abstract vi
Lista de Figuras vii
Lista de Tabelas viii
Lista de Símbolos ix
Lista de Abreviaturas e Siglas x
1 Introdução 1
1.1 Motivação 1
1.2 Objetivos 1
1.3 Estrutura do trabalho 1
2 Revisão Bibliográfica 3
2.1 Linguagem e Autômatos 3
2.2 Teoria de Controle Supervisório 4
2.3 Modbus 5
3 Materiais e Métodos 8
3.1 Materiais 8
3.1.1 Ferramentas de Software utilizadas 8
3.1.2 Ferramentas de Hardware utilizadas 12
3.2 Metodologia 14
4 Formulação do Estudo de Caso 15
4.1 Modelagem das plantas 15
4.1.1 Stack Feeder 16
4.1.2 Conveyor Belt 17
4.1.3 Exit Slide 19
4.2 Restrições 20
4.3 Supervisório 21
4.4 Detalhamento do código 22
5 Resultados 27
6 Conclusões e Trabalhos Futuros 28
7 Referências 29
Anexos 30
Anexo 1: Biblioteca MgsModbus, anexo .h: 30

Anexo 2: Biblioteca MgsModbus, anexo .cpp: 31
Apêndices 38
Apêndice 1: função main() do código: 38
Apêndice 2: função Conveyor_belt() do código: 43
Apêndice 3: função Exit_slide() do código: 43
Apêndice 4: função Exit_slide() do código: 44
Apêndice 5: função ReadCoils() do código: 44
Apêndice 6: função Stack_feeder() do código: 45
Apêndice 7: função WriteCoils() do código: 46
Apêndice 8: função atualizaDesabilitacoes() do código: 46
Apêndice 9: função atualizaSequenciasOperacionais() do código: 48
Apêndice 10: função atualizaSupervisores() do código: 48
Apêndice 11: função executaEventosControlaveis() do código: 51
Apêndice 12: função recebeRespostas() do código: 51
Apêndice 13: função select_cont_events() do código: 52
Apêndice 14: função start_cmd() do código: 53

Agradecimentos

Agradeço, primeiramente, aos meus pais, Luis Fernando e Maria Aparecida, à minha irmã, Maria Fernanda, e a todos os familiares que participaram da minha vida não somente no período de realização deste trabalho, mas também durante toda a graduação pelo apoio, carinho e suporte incondicional a mim fornecido.

À todos os professores que, ao longo do curso, dispuseram de paciência para transmitir conhecimento e sabedoria tornando este momento possível. Em especial à meu orientador prof. Dr. Marcelo Götz que não mediu esforços no auxilio da confecção do trabalho aqui apresentado.

Finalmente, agradeço a todos os meus amigos que sempre se fizeram presentes proporcionando momentos de descontração e alegria durante todos estes anos.

Resumo

O presente trabalho propõe uma infraestrutura que visa permitir a implementação em linguagem C de controladores supervisórios para plantas de manufatura em sistemas microcontrolados. A planta de manufatura virtual será simulada no computador, enquanto que o sistema supervisório será estruturado na linguagem de programação C e aplicado junto a uma placa eletrônica com microcontrolador embarcado. A fim de realizar a comunicação dentre estes sistemas, será utilizado o protocolo Modbus TCP. A estrutura de código explorada neste trabalho se baseia em diversas propostas apresentadas pelos professores Queiroz e Cury bem como no software desenvolvido pelo grupo de pesquisa em Sistemas a Eventos Discretos da Friedrich-Alexander-Universität Erlangen-Nürnberg, de acesso livre.

Palavras Chave: controle supervisório, plantas de manufatura, microcontrolador.

Abstract

The present work proposes an infrastructure that has the purpose of allowing the implementation of supervisory controllers structured for microcontrollers in C language applied over manufacturing plants. The virtual manufacturing plant will be simulated on PC located software, meanwhile the supervisory system will be structured in C programmable language and applied on an electronic board equipped with a microcontroller. The communication protocol Modbus TCP will be used for communicating all the systems involved. The code structure explored in this paper is based on several proposes presented by the professors Queiroz and Cury as well as on the software developed by the Discrete Event Systems Group of Friedrich-Alexander-Universität Erlangen-Nürnberg.

Keywords: supervisory control, manufacturing plants, microcontroller.

Lista de Figuras

Figura 1 - autômato ilustrativo de $L1$ 4
Figura 2 - modelo de mensagem Modbus TCP/IP. 6
Figura 3 - ambiente de trabalho da ferramenta DESTool. 8
Figura 4 - ambiente de trabalho da ferramenta FlexFact. 9
Figura 5 - ambiente de trabalho da ferramenta Modbus Poll. 10
Figura 6 - ambiente de trabalho da ferramenta Modbus Slave. 11
Figura 7 - placa Arduino Uno. 12
Figura 8 - placa Arduino Ethernet Shield. 13
Figura 9 - estrutura final do trabalho. 15
Figura 10 – Estudo de caso no software FlexFact. 15
Figura 11 - Representação do Stack Feeder nas ferramentas DESTool e FlexFact. 16
Figura 12 - Representação da Conveyor Belt nas ferramentas DESTool e FlexFact. 18
Figura 13 - Representação da Exit Slide nas ferramentas DESTool e FlexFact. 19
Figura 14 - restrições representadas no software DESTool. 20
Figura 15 – sistema supervisório representado no software DESTool. 21
Figura 16 - estrutura do sistema de controle supervisório. 23
Figura 17 - estrutura do código na linguagem de programação C. 25

Lista de Tabelas

Tabela 1 - funções disponíveis no protocolo Modbus. 6
Tabela 2 - características do Aduino Uno. 12
Tabela 3 - características do Arduino Ethernet Shield. 13
Tabela 4 - eventos do Stack Feeder. 17
Tabela 5 - eventos da Conveyor Belt. 18
Tabela 6 - eventos da Exit Slide. 19
Tabela 7 - tabela de eventos habilitados pelo supervisório. 21
Tabela 8 - conversão de sinais nos coils em eventos. 23

Lista de Símbolos

Hz – Hertz

MHz – Megahertz

V – Volt

Vpp – Volt pico-a-pico

Mb/s – Megabits per Second

Mbps – Megabits per Second

mV – Mili Volts

µA – Micro Ampère

mA – Mili Ampère

Ω – Ohm

A – Ampère

mA – Mili Ampère

Lista de Abreviaturas e Siglas

UFRGS – Universidade Federal do Rio Grande do Sul

TCS – Teoria de Controle Supervisório

SED – Sistemas a Eventos Discretos

TCP – Transmition Control Protocol

IP – Internet Protocol

ICSP – In-circuit Serial Programming

PWM – Pulse-Width Modulation

USB – Universal Serial Bus

SRAM – Static Random-Access Memory

EEPROM – Electrically Erasable Programmable Read-Only Memory

UDP – User Datagram Protocol

CC – Corrente Contínua

CA – Corrente Alternada

MAC – Media Access Control

1 Introdução

No mundo atual, onde a evolução tecnológica é uma realidade, a utilização de autômatos para representar o comportamento de diversos sistemas, ditos Sistemas a Eventos Discretos (SED), se tornou uma constante. Os "comportamentos SED" estão presentes nas escadas rolantes, nas portas automáticas, nos caixas de autoatendimento bancário, nos elevadores, nas sinaleiras, na automação residencial, nas linhas de montagem das fábricas; enfim, pode se dizer, de maneira geral, que autômatos são utilizados em instalações onde é necessário executar um processo de manobra, controle ou sinalização.

O ambiente industrial pode ser citado como um forte exemplo de ambiente onde muitas vezes é exigido executar processos de controle, consequentemente, autômatos se fazem presentes neste contexto.

1.1 Motivação

A principal motivação do trabalho proposto se caracteriza em agrupar ferramentas e teses disponíveis no meio acadêmico de tal forma que suas funções sejam agregadas culminando em uma plataforma experimental que permita a implementação da Teoria de Controle Supervisório (TCS) aplicada a exemplos práticos.

Agregado a isto, deseja-se com esta plataforma e exemplos práticos, permitir que os alunos que estudam esta teoria a compreendam melhor através da utilização da plataforma. Ainda, esta estrutura poderá servir de instrumento de ensaios em propostas de pesquisa em SED.

Com esta prática, espera-se, ainda, tornar este tipo de abordagem compreensível a um maior grupo de profissionais para, desta forma, motivar a utilização do modelamento por autômatos e geração de controle supervisório no contexto da indústria de manufatura.

1.2 Objetivos

A proposta de trabalho de conclusão de curso consiste em montar uma infraestrutura visando permitir a elaboração de experimentos em uma planta virtual lançando mão de ferramentas desenvolvidas pelo grupo de pesquisa em Sistemas a Eventos Discretos da Friedrich-Alexander-Universität Erlangen-Nürnberg, de acesso livre. A infraestrutura irá permitir que sejam desenvolvidos controladores supervisórios, utilizando técnicas de síntese destes controles, para plantas flexíveis de manufatura e que se possa verificar o funcionamento do controle no modelo. Uma plataforma com microprocessador será utilizada para exercer este controle sobre o modelo simulado através do protocolo de comunicação de dados Modbus e da execução em linguagem C/C++ do controle supervisório sintetizado. Assim, será possível realizar o teste do controle supervisório efetivamente implementado, e não somente a nível de modelo em autômatos.

1.3 Estrutura do trabalho

O presente trabalho está estruturado da seguinte maneira: no Capitulo 2 será apresentada uma revisão de todos os conceitos básicos necessários para a compreensão da proposta de trabalho, apresentada juntamente com suas devidas referências. No Capitulo 3 serão descritas as ferramentas de hardware e software utilizados durante a

confecção do trabalho, bem como a maneira como elas se integram de forma a compor a estrutura final do projeto. No Capitulo 4 será explicado o estudo de caso, o funcionamento e o processo de modelagem das máquinas que o compõe, além do projeto das restrições e do controle supervisório. É neste Capitulo, também, que será apresentado o código, sua estrutura lógica e suas funcionalidades. Por fim, nos Capítulos 5 e 6 são apresentados os resultados e as conclusões, respectivamente.

2 Revisão Bibliográfica

2.1 Linguagem e Autômatos

Plantas de manufatura são sistemas frequentemente representados em estudos envolvendo Sistemas a Eventos Discretos (SED), uma vez que um SED pode ser utilizado de forma a representar uma sequência de eventos. Em uma célula de trabalho de um sistema de manufatura composta por esteiras, robôs e máquinas, por exemplo, podem-se observar vários eventos, onde a sequência de ocorrência destes eventos descreve a evolução desse sistema (GÖTZ, 2013).

Uma das maneiras formais de estudar o comportamento lógico de Sistemas a Eventos Discretos é baseada em teorias de linguagem e autômatos. O ponto de partida considera que todo SED possui um conjunto de eventos *E* associado a ele. Este conjunto de eventos *E* é tido como o "alfabeto" de uma linguagem, enquanto que as sequências de eventos são consideradas como as "palavras" que integram esta mesma linguagem (CASSANDRAS e LAFORTUNE, 1999).

Conforme apresentado por (GÖTZ, 2013), define-se a linguagem *L* sobre o conjunto *E* como sendo o conjunto de sequências (de tamanho finito) de eventos contidos em *E*. Por exemplo, seja *E={a,b,g}* o conjunto de eventos. A partir deste conjunto pode-se definir a linguagem

$$L_1 = \{\varepsilon, a, abb\}$$

que consiste de três elementos somente ($|L_1| = 3$).

Ou então a linguagem

$$L_2 = \{todas\ as\ sequencias\ possíveis\ de\ tamanho\ 3\ que\ começam\ com\ o\ evento\ a\}$$

que consiste de nove elementos ($|L_2| = 9$).

Ou ainda a linguagem

$$L_3 = \{todas\ as\ sequencias\ de\ tamanho\ finito\ que\ começam\ com\ o\ evento\ a\}$$

que contem um tamanho infinito de elementos ($|L_3| = \infty$).

Um autômato, por sua vez, pode ser entendido como uma representação gráfica de uma determinada linguagem. Uma maneira apropriada de se representar um autômato é através de um grafo direcionado, chamado de diagrama de transição de estados. Um autômato de estados finitos sobre um conjunto finito de eventos *E* pode ser representado como um grafo direcionado com a propriedade de que cada nó do grafo emite um arco rotulado com cada evento contido em *E*. Os nós deste grafo são chamados de estados e representam o conjunto de estados do autômato. Há um estado especial chamado de estado inicial. Este estado é marcado por uma seta apontando diretamente para ele. Há também um conjunto de estados chamados de estados marcados. A definição dos estados marcados é uma decisão de modelagem do sistema. Um estado marcado pode representar a finalização de uma tarefa, ou então o estado de repouso de um robô que está esperando a execução da próxima tarefa, por exemplo (GÖTZ, 2013).

Acoplando os conceitos de linguagem e autômatos, finalmente, pode-se apresentar um dos autômatos existentes capaz de representar a linguagem L_1 supracitada da seguinte forma:

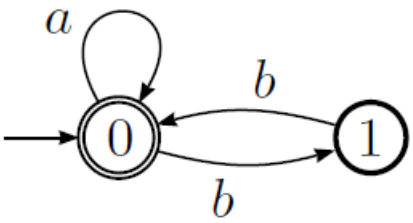

Figura 1 - autômato ilustrativo de L_1.

Fonte: Autor

A Figura 1 apresenta um autômato que possui dois estados, sendo que o estado zero é classificado como marcado e inicial. Além disso, é válido ressaltar que este autômato pode representar não somente a linguagem L_1 mas também qualquer outra linguagem que envolva os eventos nele contidos tal como, por exemplo, $L^* = \{\varepsilon, a, aa, abb, aabb, aaabb\}$.

2.2 Teoria de Controle Supervisório

A Teoria de Controle Supervisório desenvolvida inicialmente por W. M. Wonham e P. J. Ramadage nos anos 80 abrange todo o assunto que envolve um determinado SED cujo comportamento deve ser modificado através de algum controle visando atender a uma série de especificações ou restrições. Para ilustrar esta definição, assume-se um SED que é modelado por um autômato *G*, cujo espaço de estados não necessita ser finito e possui um conjunto de eventos *E.* Este autômato *G* modela o "comportamento não-controlado" do SED. A premissa é de que este comportamento não é satisfatório e deve ser modificado através de controle. Esta modificação de comportamento pode ser compreendida como a restrição do funcionamento do autômato a um subconjunto de $L(G)$. Visando alterar o comportamento de *G*, é introduzido um supervisor denotado como *S.* Nota-se que nesta análise foi separado a "planta" *G* do "controlador" (ou supervisório) *S* assim como na teoria de controle clássica (CASSANDRAS e LAFORTUNE, 1999).

O supervisório *S* age de forma a observar todos os eventos do sistema, inferindo, deste modo, em que estado o sistema se encontra. Por esta razão, este controle recebe o nome de supervisório. A ação de controle se dá em desabilitar eventos, dentre aqueles que poderiam ser executados no estado em que o sistema se encontra, para, então, evitar que o sistema entre em estados indesejados ou execute sequências de eventos não permitidas. O supervisório é, portanto, um controle que age de maneira dinâmica na medida em que observa os eventos executados pelo SED sob controle, e eventualmente desabilita certos eventos. O grau de complexidade na definição de um controle supervisório se dá pelo motivo de que nem todos os eventos de um SED são observados,

assim como nem todos os eventos de um SED podem ser controlados. Um evento não controlável é aquele que não pode ser desabilitado pelo supervisório (GÖTZ, 2013).

2.3 Modbus

O Modbus é um protocolo de mensagem que integra a camada de aplicação do modelo OSI utilizado para comunicação entre dispositivos mestre-escravo/cliente-servidor. Inicialmente desenvolvido pela fabricante de equipamentos Modicon na década de 70, o protocolo teve seus direitos transferidos para a Modbus Organization em 2004 depois que a empresa Modicon foi adquirida pela Schneider Eletric. O protocolo Modbus é um dos mais antigos e mais utilizados protocolos de comunicação, sendo aplicado em uma vasta gama de sistemas de automatizados tais como: instrumentos e equipamentos de laboratório, automação residencial, naval e industrial (MODBUS APPLICATION PROTOCOL SPECIFICATION, 2012).

O protocolo de comunicação Modbus é um dos mais utilizados ainda nos dias de hoje, muito devido à sua simplicidade e facilidade de implementação, podendo ser utilizado em variados padrões de meio físico, como: EIA/TIA-232-E, EIA-422, EIA/TIA-485-A, radio, fibra, Ethernet TCP/IP (Modbus TCP). Este último padrão citado será o empregado no presente trabalho uma vez que o software FlexFact, utilizado para simular a planta de manufatura, se comunica através de Modbus TCP.

O funcionamento básico do protocolo Modbus pode ser descrito como segue: a estação mestre, ou cliente do ponto de vista cliente/servidor, inicia a comunicação solicitando que os escravos (servidores) enviem seus dados. Os escravos, por sua vez, recebem a requisição do mestre e retornam os dados solicitados. O mestre Modbus manda junto ao frame de dados um código referente à função que se deseja executar junto ao escravo (leitura, escrita, entre outros). No protocolo Modbus cada função (denotada pela sigla FC, do inglês, *Function Code*) é utilizada para acessar um tipo específico de dado.

No presente trabalho, no entanto, só será necessário o uso de duas funções presentes na Tabela 1: FC 01- leitura de bloco de bits do tipo coil e FC 15- escrita em bloco de bits do tipo coil. Isto se dá uma vez que as máquinas integrantes da planta simulada no software FlexFact (explicado posteriormente no texto) utilizam coils para indicar e modificar, caso requisitado, o status dos equipamentos que as integram, tais como sensores e motores.

Tabela 1 - funções disponíveis no protocolo Modbus.

Fonte: EMBARCADOS (2014).

Código da função	Descrição
1	Leitura de bloco de bits do tipo coil (saída discreta).
2	Leitura de bloco de bits do tipo entradas discretas.
3	Leitura de bloco de registradores do tipo holding.
4	Leitura de bloco de registradores do tipo input.
5	Escrita de um único bit do tipo coil (saída discreta).
6	Escrita em um único registrador do tipo holding.
7	Ler o conteúdo de 8 estados de exceção.
8	Prover uma série de testes para verificação da comunicação e erros internos.
11	Obter o contador de eventos.
12	Obter um relatório de eventos.
15	Escrita em bloco de bits do tipo coil (saída discreta).
16	Escrita em bloco de registradores do tipo holding.
17	Ler algumas informações do dispositivo.
20	Ler informações de um arquivo.
21	Escrever informações de um arquivo.
22	Modificar o conteúdo de registradores de espera através de operações lógicas.
23	Combina ler e escrever em registradores numa única transação.
24	Ler o conteúdo da fila FIFO de registradores.
43	Identificação do modelo do dispositivo.

Na especificação do protocolo são definidos dois modos de transmissão: ASCII e RTU. Os modos definem como são transmitidos os bytes da mensagem, e como a informação da mensagem será empacotada na mensagem e descompactada. Porém, como neste trabalho somente será implementado o Modbus TCP, este é o único padrão que terá seu funcionamento especificado.

Modbus TCP, portanto, é uma implementação do protocolo Modbus baseado no protocolo TCP/IP. O Modbus utiliza a pilha TCP/IP para comunicação e adiciona ao quadro Modbus um cabeçalho específico chamado MBAP (Modbus Application Protocol). O modelo de mensagem TCP/IP pode ser observado na Figura 2.

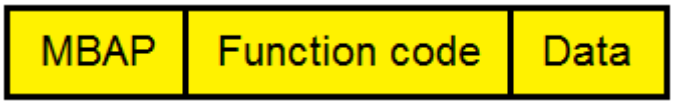

Figura 2 - modelo de mensagem Modbus TCP/IP.

Fonte: EMBARCADOS (2014).

Objetivando lançar mão do uso do padrão Modbus TCP, se fez necessário utilizar uma placa eletrônica com um microcontrolador embarcado chamada de Arduino (ARDUINO, 2014a) juntamente com um Shield Ethernet (ARDUINO, 2014b) desenvolvido pela mesma empresa. Deste modo, se torna possível exercer a comunicação entre o microcontrolador, onde o sistema supervisório se encontrará, e a planta de manufatura simulada no software FlexFact localizado no computador. Porém, antes que essa comunicação pudesse ser realizada, foi preciso buscar uma biblioteca Modbus Mestre (cliente) que dispusesse suporte ao Shield utilizado, bem como ao padrão de comunicação escolhido. A biblioteca encontrada para exercer tal função é chamada de MgsModbus (MYARDUINOPROJECTS, 2013), de uso livre. Entretanto, esta biblioteca, por estar em um estágio bastante incipiente, necessitou de um profundo estudo de seu código e de sua estrutura de funcionamento, para que modificações fossem realizadas e as funcionalidades do Modbus TCP Mestre fossem executadas.

3 Materiais e Métodos

3.1 Materiais

3.1.1 Ferramentas de Software utilizadas

3.1.1.a DESTool

DESTool (FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG, 2014a) é uma ferramenta voltada para a modelagem, síntese e análise de sistemas a eventos discretos, utilizando o conceito de autômatos. Tecnicamente, esta ferramenta apenas representa graficamente a biblioteca de sistemas a eventos discretos chamada de libFAUDES; ou seja, o usuário utiliza o DESTool para estruturar o problema de interesse visualmente, de forma a criar um projeto composto por funções e variáveis contidas na biblioteca faudes.

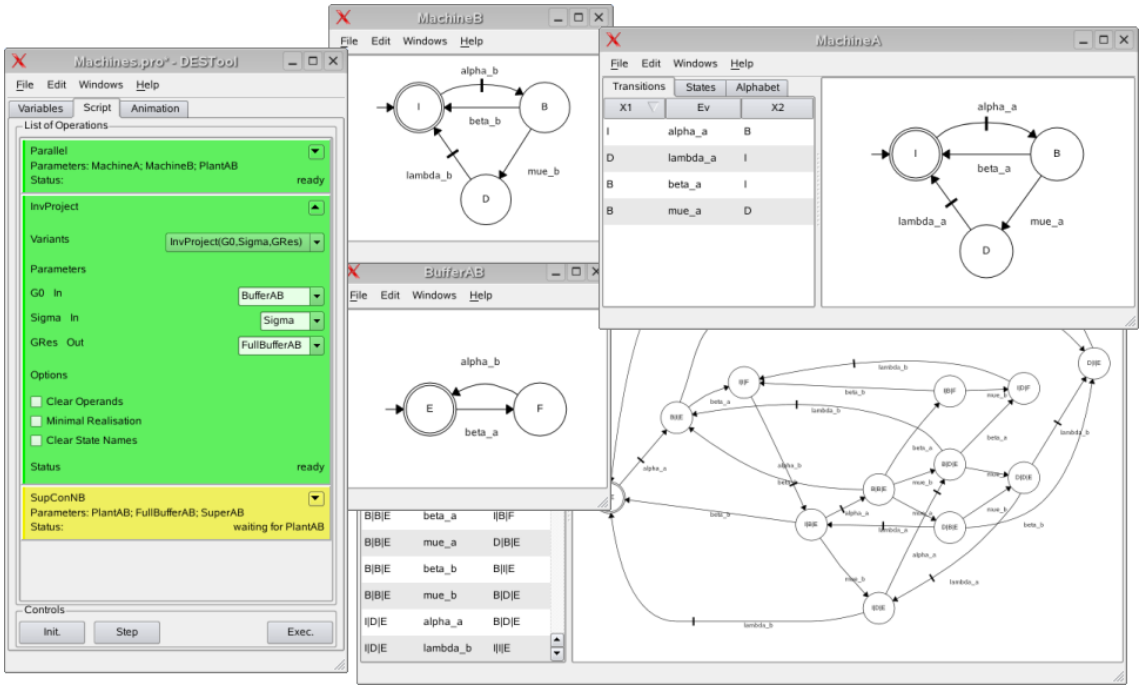

Figura 3 - ambiente de trabalho da ferramenta DESTool.

Fonte: FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG (2014a).

No projeto realizado, DESTool será utilizado de maneira a auxiliar na síntese do controle supervisório que irá atuar sobre o modelo da planta. Objetivando executar esta tarefa, portanto, é lançado mão de outro atributo importante dessa ferramenta: a possibilidade de se comunicar com a planta através de protocolos de comunicação como Modbus/TCP e Simplenet.

Como será abordado na seção seguinte, a planta será estruturada e simulada em outro software chamado FlexFact; ou seja, resumidamente, a ferramenta DESTool, em um primeiro momento, conterá o controle que deverá atuar no modelo da planta presente

no FlexFact utilizando, para tanto, um protocolo de comunicação disponível em ambas ferramentas.

3.1.1.b FlexFact

O FlexFact (FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG, 2014b) é um simulador de plantas de manufatura que imita o comportamento dinâmico de vários componentes eletromecânicos como esteiras motorizadas e máquinas de processamento para, deste modo, formar um sistema flexível de manufatura. A ferramenta permite que o usuário escolha e posicione os componentes que irão compor o seu layout de fábrica personalizado.

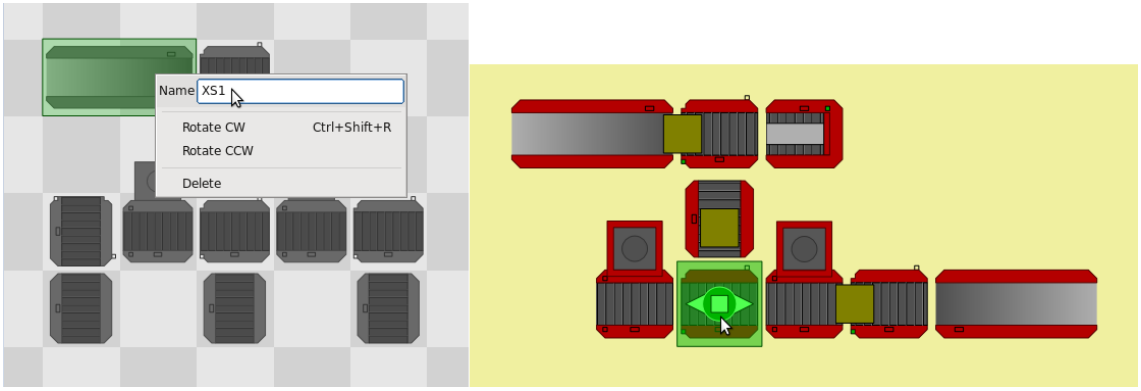

Figura 4 - ambiente de trabalho da ferramenta FlexFact.

Fonte: FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG (2014b).

É com o auxilio desta ferramenta que o modelo da planta simulada será sintetizado. Uma característica importante que valida o software como um simulador fiel é que a planta simulada exige as mesmas conexões, em termos de cabeamento, para comunicação e aplicação do controle, do que uma planta real; isto é, o trabalho que será realizado neste aspecto é muito semelhante ao que seria realizado se um projeto similar fosse realizado em campo.

Será sobre esta planta simulada no FlexFact, portanto, que o controle irá atuar e enviar sinais por meio do protocolo de comunicação Modbus/TCP onde esta será o chamado escravo enquanto que, na versão final do projeto, o Arduino representará o mestre.

3.1.1.c Modbus Poll

O software Modbus Poll (MODBUSTOOLS, 2014a) é um simulador de mestre Modbus projetado para o auxilio do desenvolvimento de estruturas que utilizam o protocolo de comunicação Modbus. Com ele é possível ler e escrever em registradores de maneira independente e prática, bastando fornecer apenas dados básicos do escravo.

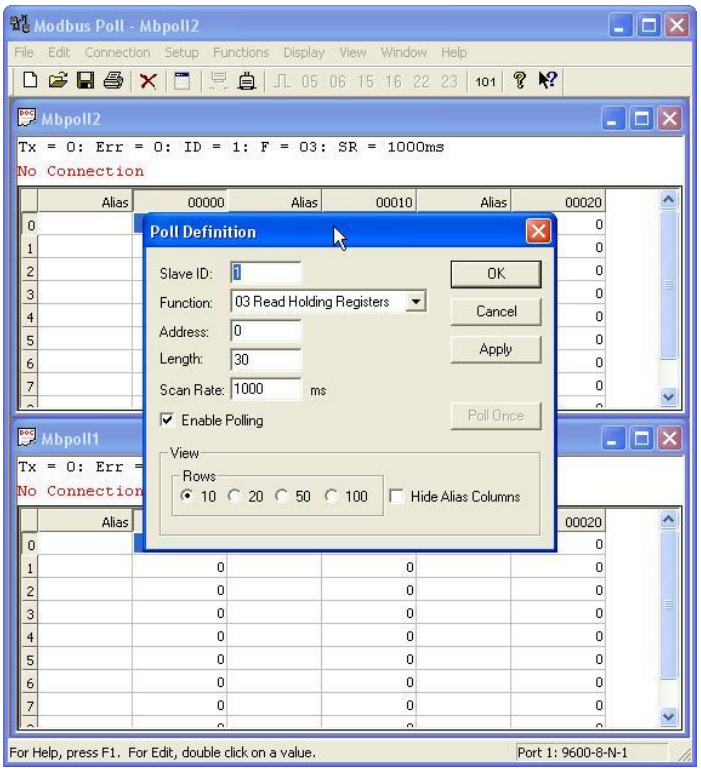

Figura 5 - ambiente de trabalho da ferramenta Modbus Poll.

Fonte: MODBUSTOOLS (2014a).

Neste trabalho, esta ferramenta é utilizada na fase de desenvolvimento da comunicação entre a planta simulada e a interface que gera os sinais de controle. Com seu auxilio é possível verificar se os sinais que transitam entre estes dois componentes estão de acordo com o esperado em termos de endereço e nível lógico.

3.1.1.d Modbus Slave

O software Modbus Slave (MODBUSTOOLS, 2014b) é um simulador de escravo Modbus projetado para o auxilio do desenvolvimento de estruturas que utilizam o protocolo de comunicação Modbus. Com ele é possível ler e escrever em registradores de maneira independente e prática, bastando fornecer apenas dados básicos como a porta utilizada (no caso Modbus TCP).

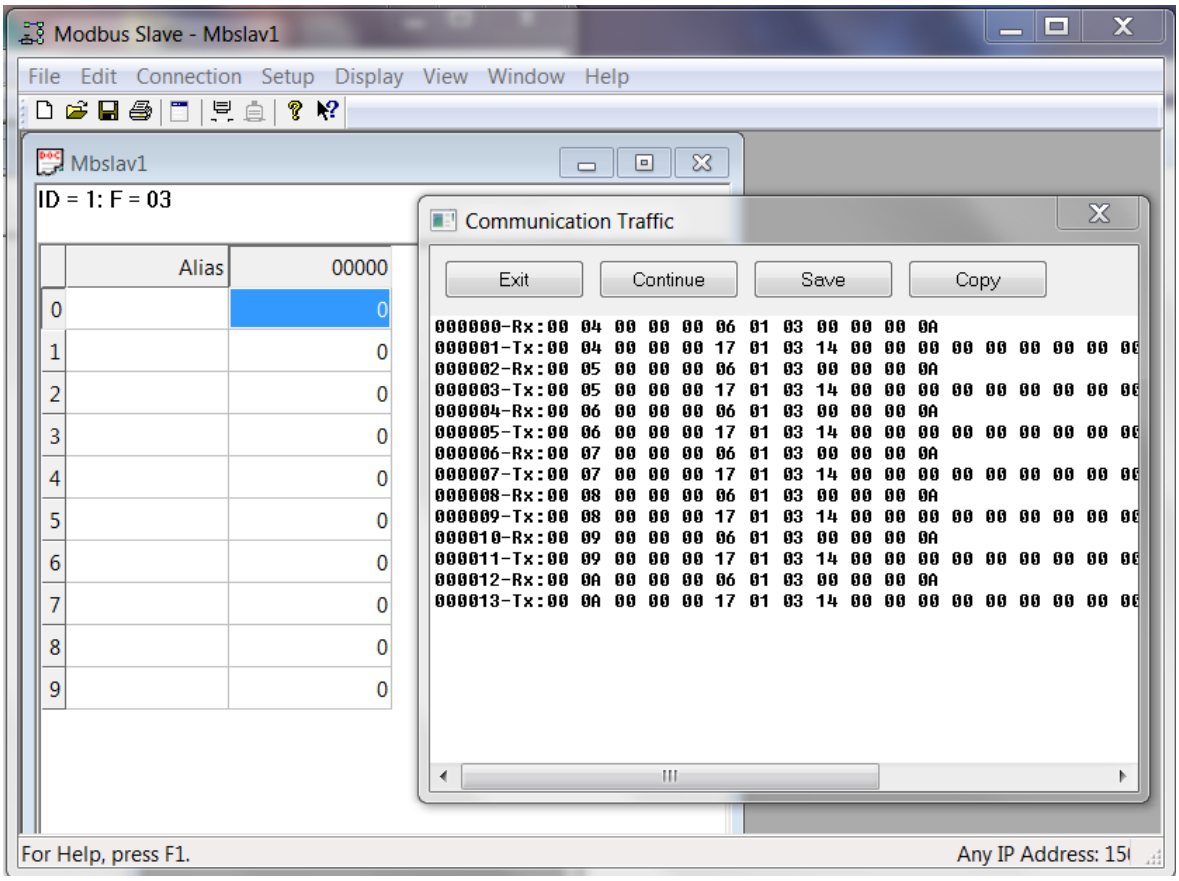

Figura 6 - ambiente de trabalho da ferramenta Modbus Slave.

Fonte: Autor.

Neste trabalho, esta ferramenta é utilizada na fase de desenvolvimento da comunicação entre o Arduino e a planta simulada no software FlexFact. Com seu auxilio é possível verificar se as mensagens estão estruturadas de acordo com o protocolo Modbus e, também, se os sinais que transitam entre estes dois componentes estão de acordo com o esperado em termos de endereço e nível lógico.

3.1.2 Ferramentas de Hardware utilizadas

3.1.2.a Arduino

O Arduino (ARDUINO, 2014a) é uma plataforma eletrônica com microcontrolador embarcado que permite ao usuário as mais diversas aplicações no âmbito dos projetos que envolvem programação aliados a circuitos elétricos. Neste projeto será utilizada a placa chama de Arduino Uno.

O Arduino Uno é uma placa eletrônica que possui um microcontrolador ATmega328 embarcado. Esta placa é composta por 14 pinos de entrada/saída (6 podem ser utilizados com PWM), 6 pinos de entrada analógica, um cristal de 16 MHz, conexão USB, conexão para alimentação externa, conector ICSP e um botão de reset.

Tabela 2 - características do Aduino Uno.

Fonte: Autor

Microcontrolador	ATmega328
Tensão de operação	5V
Tensão de entrada (recomendada)	7 – 12V
Tensão de entrada (limites)	6 – 20V
Pinos de I/O digital	14 (6 dos quais podem ser PWM)
Pinos de entrada analógica	6
Corrente continua por pino de I/O	40 mA
Corrente continua para o pino de 3.3V	50 mA
Memória Flash	32 KB
SRAM	2 KB
EEPROM	1 KB
Clock	16 MHz

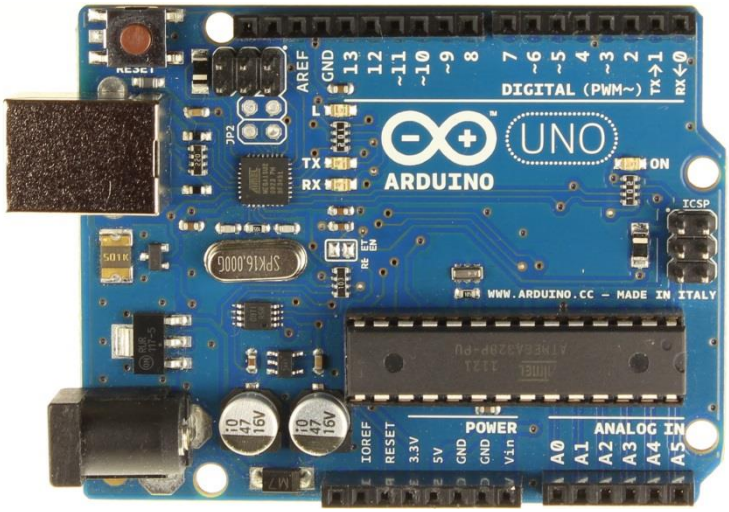

Figura 7 - placa Arduino Uno.

Fonte: ARDUINO (2014a).

Neste projeto, o Arduino Uno será utilizado como plataforma para a programação do controle que irá atuar sobre a planta simulada no software FlexFact. Visando cumprir essa tarefa, será utilizado o protocolo de comunicação Modbus TCP onde, neste contexto, ele será o mestre e a planta, o escravo. Portanto, para permitir que essa comunicação mestre-escravo aconteça, é necessário o uso de um acessório chamado de Shield junto ao Arduino. Na seção seguinte deste trabalho serão abordadas as principais características e funções deste acessório.

3.1.2.b Shield Ethernet

O Shield Ethernet (ARDUINO, 2014b) para o Arduino (ou *Arduino Ethernet Shield*) é um acessório que permite conectar o Arduino Uno à internet rapidamente. Basta acoplar o módulo sobre a placa do Arduino e conectar um cabo RJ45 à rede. Este acessório é baseado no chip ethernet Wiznet W5100 que fornece um *network (IP) stack* suportando tanto TCP quanto UDP.

Uma vez estabelecidas as conexões necessárias, é preciso ainda programar o Arduino de modo que ele atue com a função desejada de mestre em uma rede Modbus TCP; para tanto, são utilizadas funções e comandos contidos na biblioteca Ethernet disponível no compilador da ferramenta.

Tabela 3 - características do Arduino Ethernet Shield.

Fonte: Autor.

Baixo ruído e *ripple* da tensão de saída (100mVpp)
Limites de tensão de entrada: 36 – 57V
Proteção de sobrecarga e curto-circuito
Tensão de saída: 9V
Alta eficiência do conversor CC/CA: 50% – 75%
1500V de isolação entre entrada e saída

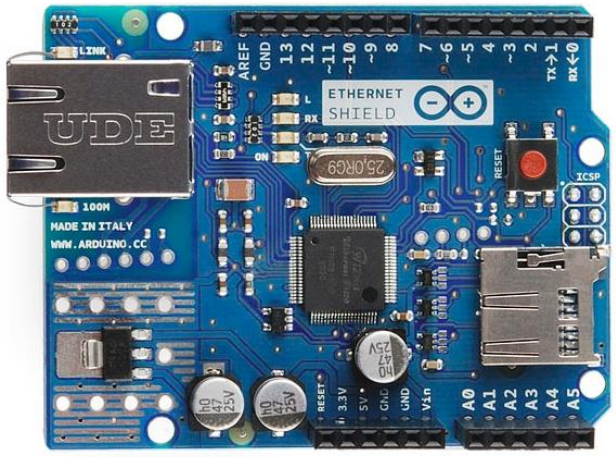

Figura 8 - placa Arduino Ethernet Shield.

Fonte: ARDUINO (2014b).

Como abordado anteriormente, o Arduino Ethernet Shield será utilizado neste trabalho como meio para possibilitar a comunicação entre o Arduino Uno, onde o controle está programado, e o computador, onde a planta está sendo simulada no software FlexFact. O protocolo de comunicação utilizado para realizar esta tarefa é o Modbus TCP, cujos detalhes serão demonstrados na seção correspondente deste trabalho.

3.2 Metodologia

Nesta seção será apresentado como as ferramentas de hardware e software irão interagir de forma a compor a estrutura base do projeto. Inicialmente, foi realizada a estrutura de comunicação Modbus TCP, utilizando o software Modbus Slave juntamente com o conjunto Arduino- Ethernet Shield e a biblioteca MgsModbus. Paralelo a esta atividade, foram desenvolvidos os modelos das máquinas que compõe o estudo de caso, bem como o controle supervisório através do software DESTool. Uma vez que a comunicação Modbus TCP se encontrava em funcionamento, foi iniciado o processo de estruturação dos modelos das máquinas em linguagem de programação C.

O processo de estruturação do código foi iniciado com um exemplo mais simples que representava apenas um possível comportamento de uma planta genérica confeccionada apenas para auxiliar na compreensão do método de programação. Após o conhecimento sobre a estrutura de programação do código ter sido obtido, foi possível iniciar a estruturação do código que seria utilizado no trabalho. A ferramenta de software utilizada para auxiliar esta etapa do trabalho é fornecida pela própria empresa distribuidora da plataforma microcontrolada (Arduino).

Durante esta fase, foi muito utilizada a comunicação através do protocolo Modbus TCP entre Arduino e Modbus Slave, pois era de interesse saber não somente se a comunicação estava funcionando mas também o que estava sendo comunicado. Em sequência, com as mensagens no formato Modbus validadas, foi iniciado o processo final de confecção da estrutura que consta em prover a comunicação entre Arduino e FlexFact diretamente, sem interferência de qualquer outra ferramenta. O estado final obtido do trabalho consiste, portanto, de dois componentes: um microcontrolador (do Arduino), executando o código que contém o controle supervisório projetado, e a planta virtual do estudo de caso, simulada no software FlexFact (no computador). Esses dois componentes principais, por fim, se comunicam através do protocolo de comunicação Modbus TCP. A estrutura final do trabalho pode ser observada na Figura 9.

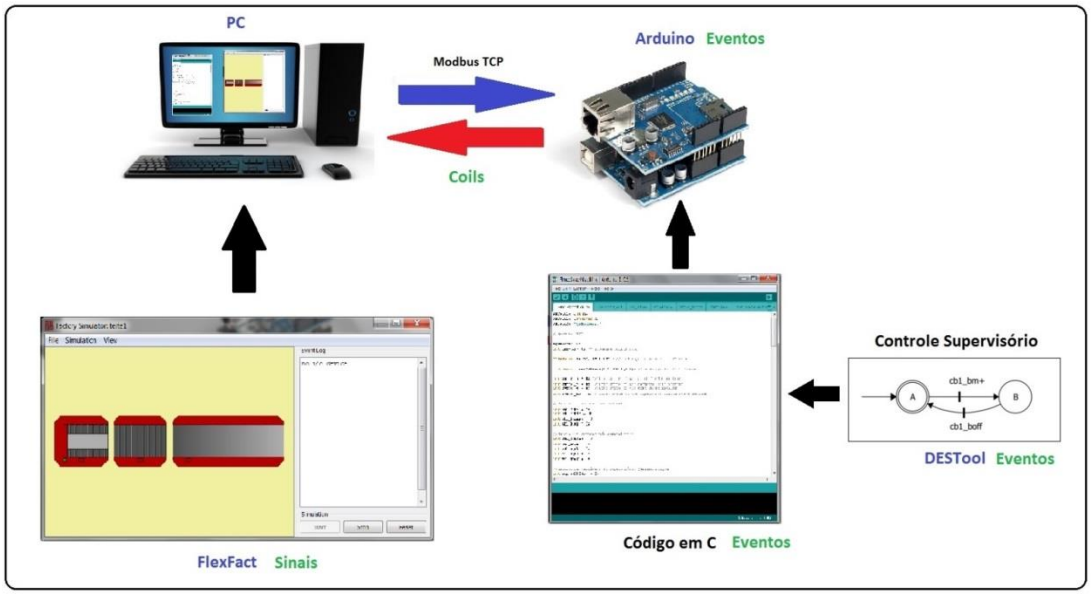

Figura 9 - estrutura final do trabalho.

Fonte: Autor.

4 Formulação do Estudo de Caso

Nesta seção será apresentada a maneira como as ferramentas serão utilizadas visando alcançar o objetivo final do trabalho. Além disso, também será apresentada a síntese do controle supervisório, bem como a modelagem das máquinas que compõe a planta de manufatura do estudo de caso.

4.1 Modelagem das plantas

A metodologia utilizada neste trabalho para a síntese do supervisório segue a mesma proposta por Ramadge e Wonham (1989) constituída de três etapas: modelagem da planta a ser controlada, modelagem das especificações desejadas para a planta e síntese de uma lógica de controle ótima. Neste trabalho a ferramenta DESTool é utilizada como ferramenta de suporte para síntese das restrições, plantas e supervisores.

Para representar o sistema de manufatura presente na Figura 10 em termos de eventos e estados, foi realizado o modelo de cada máquina integrante da planta de maneira separada. Na sequência serão descritos os modelos realizados bem como o funcionamento dessas máquinas.

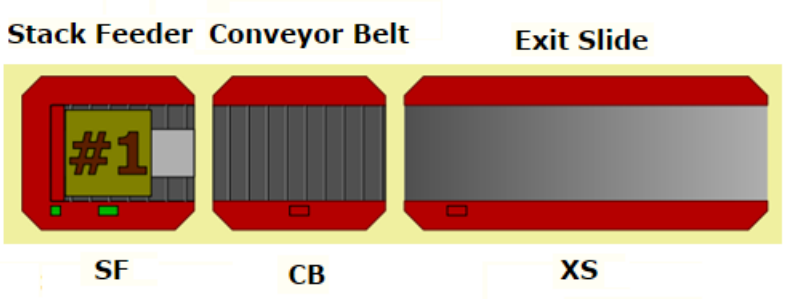

Figura 10 – Estudo de caso no software FlexFact.

Fonte: Autor.

4.1.1 Stack Feeder

A máquina chamada de Stack Feeder tem a função básica de alimentar a linha de produção com novas peças a serem trabalhadas. É composta por basicamente por três componentes: motor, sensor de peça e sensor de feeder. O motor é o componente responsável por empurrar a peça para a próxima etapa na linha de produção, representado na Figura 11 por um elemento retangular vermelho logo acima do ponto em verde claro.

O sensor de peça por sua vez, tem função de informar a existência de uma nova peça na pilha (no Stack) pronta para ser passada adiante. Na Figura 11 este é representado por uma pequena moldura retangular em preto ao lado do ponto em verde claro. O sensor de feeder, finalmente, é um sensor de presença que cumpre papel semelhante ao de peça com a diferença que o objeto detectado é, agora, o feeder. Ou seja, o componente fixado ao motor que executa a força necessária para empurrar a peça do stack para a próxima máquina na linha de produção. Na Figura 11 este está representado como um ponto verde claro.

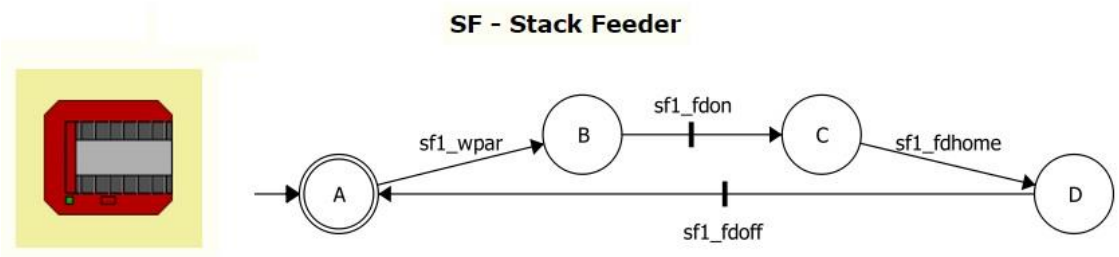

Figura 11 - Representação do Stack Feeder nas ferramentas DESTool e FlexFact.

Fonte: Autor.

A Figura 11 ilustra a máquina Stack Feeder representada nos dois principais softwares utilizados no presente trabalho: DESTool e FlexFact. Na ferramenta FlexFact é possível simular componentes reais da planta de manufatura enquanto que a DESTool possibilita representar tais máquinas em termos de estados e eventos da forma que mais interesse ao usuário.

Para modelar a máquina em questão, portanto, foi utilizado o conjunto de eventos pré-determinados pelo desenvolvedor do software. Este conjunto de eventos é composto por eventos controláveis e eventos não controláveis como disposto na Tabela 4.

Tabela 4 - eventos do Stack Feeder.

Fonte: Autor.

Stack Feeder	Status da máquina	Eventos Controláveis	Eventos não controláveis
Motor de feeder	Ligado Desligado	sf_fdon sf_fdoff	—
Sensor de peça	Chegada da peça Partida da peça	—	sf_wpar sf_wplv
Sensor de feeder	Feeder em posição (positivo)	—	sf_fdhome

Pode-se observar que o modelo da máquina apresentado na Figura 11 não possui todos os eventos dispostos na Tabela 4. Isto ocorre, pois o interesse da modelagem da máquina é focado apenas em representar o comportamento básico desta, sem nenhum tipo de restrição ou ordem de funcionamento. Estas restrições quanto ao funcionamento serão desenvolvidas pelo supervisório ainda a ser realizado.

Uma breve explicação sobre o funcionamento do modelo representado na Figura 11 pode ser descrito como segue: no estado “A” a máquina se encontra com todos os seus elementos em repouso e aguarda apenas uma peça ser posicionada de maneira com que o sensor de peça detecte sua presença. Uma vez que a peça é inserida, o evento “sf_wpar” ocorre e o motor que a irá empurrar adiante é ligado, ou seja, “sf_fdon” também acontece. O evento que implica no desligamento do motor (sf_fdoff) somente ocorre quando o sensor de posicionamento de feeder (elemento acoplado ao motor que exerce a força sobre a peça) for acionado, em outras palavras, quando sf_fdhome sucede. É neste instante que o autômato retorna ao seu estado inicial “A” para, se necessário, executar outro ciclo.

4.1.2 Conveyor Belt

A máquina chamada de Conveyor Belt tem a função de transportar a peça de uma máquina para outra na linha de produção. Esta máquina possibilita ao usuário o transporte da peça em ambos os sentidos bem como sua parada em qualquer posição intermediária da esteira.

A Conveyor Belt é formada basicamente por duas partes, são elas: motor (conjunto esteira acoplada ao motor) e sensor de presença de peça. O motor, juntamente com a esteira, tem a função de realizar o movimento da peça no sentido desejado enquanto que o sensor de presença é utilizado para avisar o usuário se a peça passou por determinado ponto ao longo da esteira. Na Figura 12 o conjunto motor-esteira é representado pelos elementos retangulares em cinza que formam uma esteira; o sensor de peça, por sua vez, é ilustrado como uma pequena moldura retangular na cor preta localizada no centro da máquina perto da borda inferior. Quando na presença de alguma peça o sensor é ativado e a moldura assume a cor verde claro como pode ser observado na Figura 12.

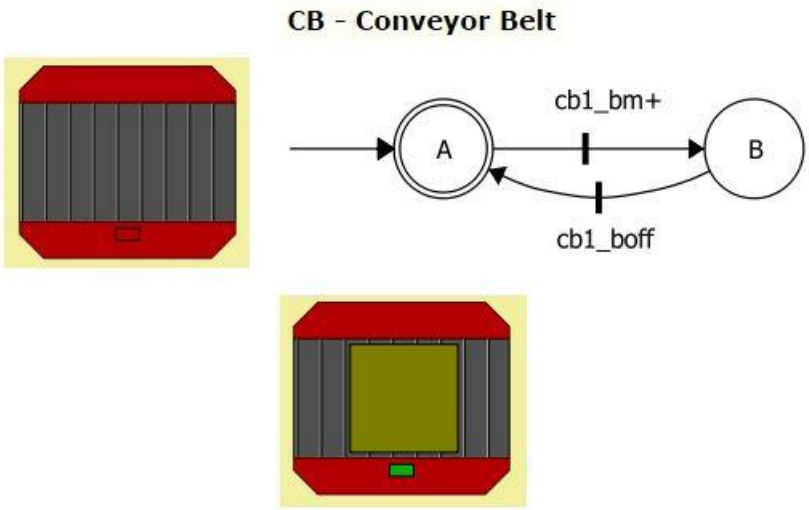

Figura 12 - Representação da Conveyor Belt nas ferramentas DESTool e FlexFact.

Fonte: Autor.

Uma vez mais foram utilizados os eventos pré-determinados para modelar a máquina na ferramenta DESTool. O conjunto de eventos disponíveis para utilização está disposto na Tabela 5.

Tabela 5 - eventos da Conveyor Belt.

Fonte: Autor.

Conveyor Belt	Status da máquina	Eventos Controláveis	Eventos não controláveis
Motor	Ligado para leste Ligado para oeste Desligado	cb_bm+ cb_bm- cb_boff	—
Sensor de peça	Chegada da peça Partida da peça	—	cb_wpar cb_wplv

O modelo idealizado para a máquina não utiliza todos os eventos disponíveis, pois o conceito neste instante é, novamente, apenas representar o funcionamento mais rudimentar e básico da máquina. Portanto, o autômato presente na Figura 12 pode ter o seu funcionamento brevemente explicado como segue: no estado “A” a máquina se encontra em repouso apenas esperando o usuário disparar o comando de ligar a esteira na direção leste que irá ser refletido pelo evento “cb_bm+”. Uma vez que isto ocorreu, a máquina estará no estado “B” apenas aguardando um novo comando para realizar o desligamento do motor, indicado pelo evento “cb_boff”, quando a máquina retornará ao estado inicial “A”. Ou seja, resumidamente, a máquina de estados que representa a Conveyor Belt simplesmente modela uma máquina que liga e desliga em função de eventos controláveis.

4.1.3 Exit Slide

A máquina chamada de Exit Slide é o componente mais simples do estudo de caso apresentado neste trabalho uma vez que possui apenas a função de acumular peças no final da linha de produção. A Exit Slide é composta apenas por um componente capaz de fornecer sinais ao usuário: o sensor de presença. Este sensor apenas detecta quando uma peça passa por ele, gerando um evento quando da chegada da peça, e outro quando peça deixa de sensibilizar o sensor. A peça desliza até o final da máquina sob o efeito da gravidade. A esteira em que as peças deslizam tem a capacidade se armazenar no máximo quatro peças de cada vez como mostrado na Figura 13.

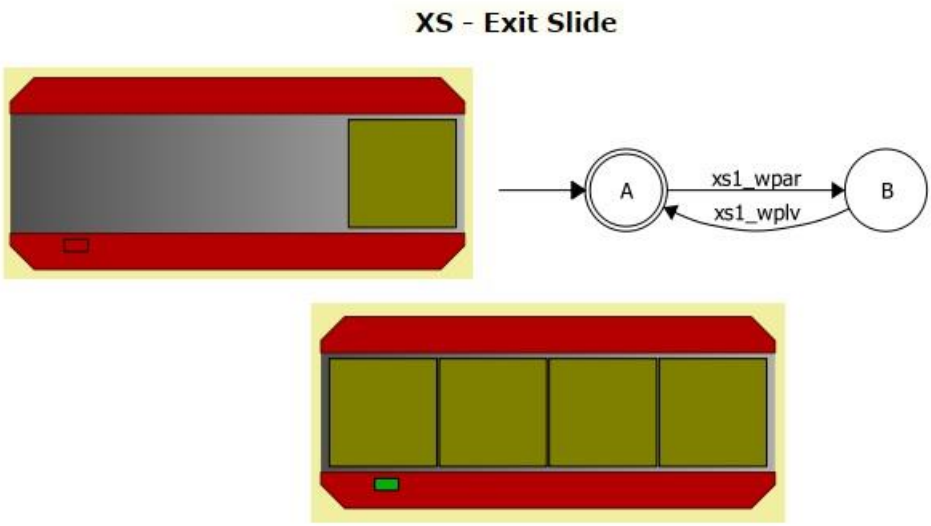

Figura 13 - Representação da Exit Slide nas ferramentas DESTool e FlexFact.

Fonte: Autor.

O conjunto de eventos disponíveis para modelar a máquina está disposto na Tabela 6.

Tabela 6 - eventos da Exit Slide.

Fonte: Autor.

Stack Feeder	Status da máquina	Eventos Controláveis	Eventos não controláveis
Sensor de peça	Chegada da peça Partida da peça	—	xs_wpar xs_wplv

O modelo da máquina representado no autômato observado na Figura 13 pode ter o seu o funcionamento descrito como segue: inicialmente no estado “A” a máquina aguarda até o instante em que o sensor detecta presença de uma peça quando muda para o estado “B” e neste ficará até que esta mesma peça desobstrua o sensor ocasionando a mudança para o estado inicial “A”.

4.2 Restrições

Na seção anterior foram apresentadas as máquinas que compõem a planta de manufatura e seus respectivos modelos em autômatos, bem como seu funcionamento, sinais e eventos. Porém, os modelos idealizados inicialmente apenas representam o comportamento básico destas máquinas, tais como ligar, desligar e acionamento de sensores. Nesta seção, entretanto, será explicado o desenvolvimento das restrições e especificações de funcionamento do sistema, os quais irão atuar juntamente com os modelos iniciais a fim de garantir o funcionamento da planta da maneira desejada pelo usuário. É importante salientar, também, que as restrições, juntamente com o modelo da planta, irão ser utilizadas para síntese do controle supervisório que será descrito no decorrer deste trabalho.

O estudo de caso abordado na Figura 10 apresenta poucos componentes e, por consequência, são necessárias apenas duas restrições para que a planta funcione de uma maneira considerada correta. Essas restrições estão presentes na Figura 14.

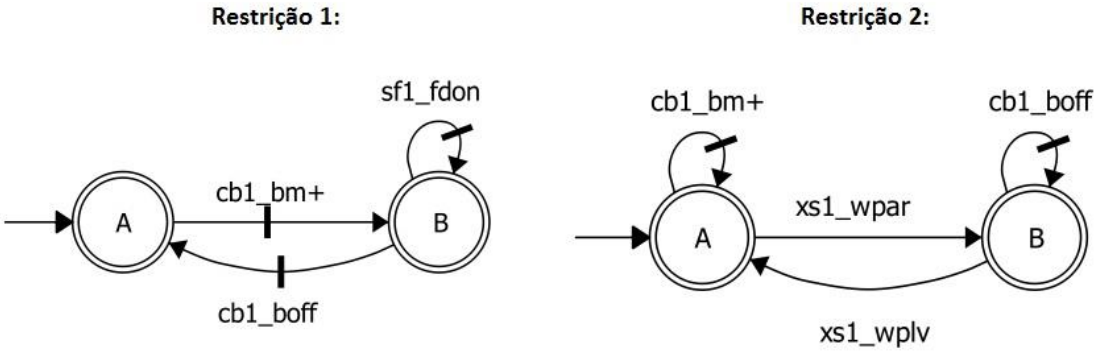

Figura 14 - restrições representadas no software DESTool.

Fonte: Autor.

A Restrição 1 foi projetada com o principio de permitir que o motor do Stack Feeder apenas possa ser ligado uma vez que a Conveyor Belt já tenha dado início ao deslocamento da sua esteira no sentindo oeste-leste. Com este principio descrito em mente foi realizado o autômato que pode ser observado na Figura 14. A Restrição 2, por sua vez, foi criada com o intuito de estabelecer a condição de que toda a vez que uma peça for detectada pelo sensor de presença da Exit Slide, a esteira da Conveyor Belt deve ser desligada e só volte a ser ligada quando este mesmo sensor for desobstruído. Assim como a restrição anterior, esta também pode ser observada no autômato representado na Figura 14.

4.3 Supervisório

Após modelar as máquinas e as restrições, é possível sintetizar o supervisório propriamente dito. Como mencionado anteriormente na seção de revisão bibliográfica, o supervisório *S* age de forma a observar todos os eventos do sistema, inferindo, deste modo, em que estado o sistema se encontra. A ação de controle se dá em desabilitar eventos no estado em que o sistema se encontra, desta maneira pode-se interpretar a Figura 15 do supervisório sintetizado para o estudo de caso como a Tabela 7 onde o símbolo "✓" significa que o evento em questão está habilitado, enquanto "—" sinaliza evento desabilitado.

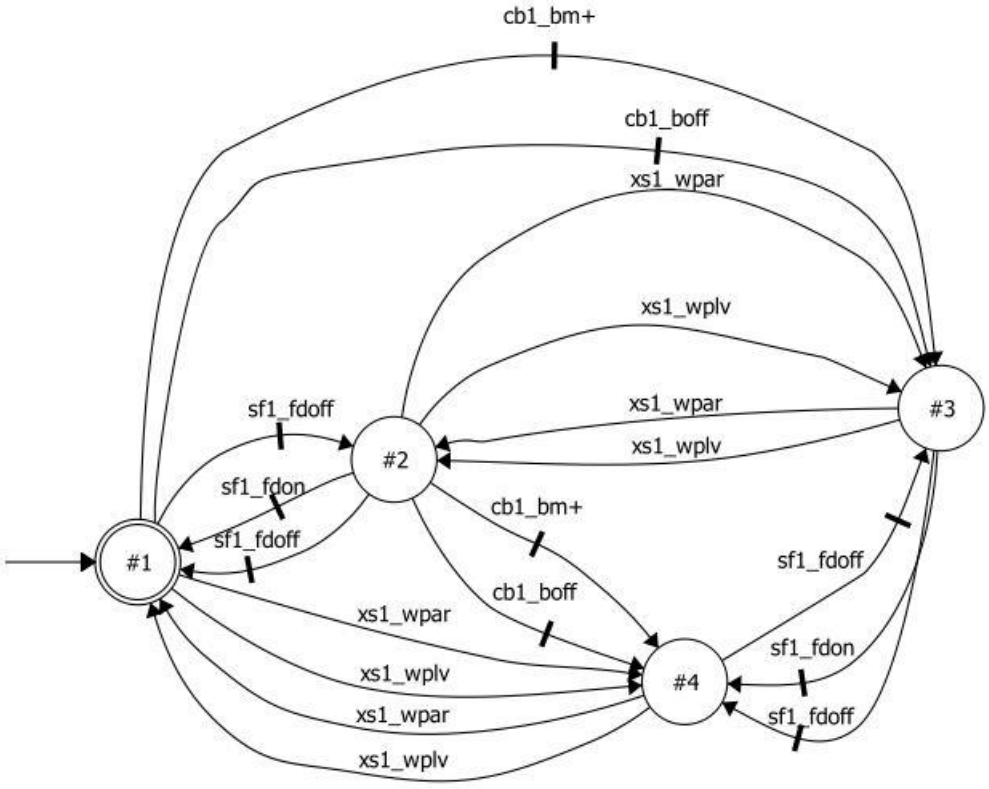

Figura 15 – sistema supervisório representado no software DESTool.

Fonte: Autor.

Tabela 7 - tabela de eventos habilitados pelo supervisório.

Fonte: Autor.

Eventos Controláveis	Estado 1	Estado 2	Estado 3	Estado 4
cb1_bm+	✓	✓	—	—
cb1_boff	✓	✓	—	—
sf1_fdon	—	✓	✓	—
sf1_fdoff	✓	✓	✓	✓

4.4 Detalhamento do código

Nesta seção será aprofundado o estudo do código utilizado no presente trabalho, abordando assuntos relacionados à sua estrutura, bem como todas as funções que o integram. Antes de partir para esta explicação, no entanto, é necessário deixar clara a diferença entre sinais e eventos.

A confusão entre o significado de sinais e eventos é um erro frequente que pode surgir durante o processo de compreensão da estrutura básica do presente trabalho. Isto ocorre uma vez que a ferramenta FlexFact, que simula a planta de manufatura física, se comunica utilizando sinais. Ou seja, cada máquina possui seu próprio conjunto de sinais cuja quantidade varia conforme a sua complexidade, número de motores, sensores e funcionalidades. Estes sinais, por sua vez, modificam os valores dos chamados coils que serão utilizados para o processo de escrita e leitura no protocolo Modbus.

De forma resumida, portanto, as máquinas simuladas no software FlexFact emitem e recebem sinais (sensores e atuadores, respectivamente) que modificam o valor dos respectivos coils utilizados no protocolo Modbus. É importante ter em mente que nesta seção, até este instante, somente foi abordado o que diz respeito a sinais. O tópico eventos surge no instante em que a outra ferramenta chamada de DESTool é empregada.

A DESTool é a ferramenta que auxilia na síntese do controle supervisório, dos modelos das máquinas e das restrições observadas. Como trabalha com sistemas a eventos discretos, somente estados e eventos são utilizados durante o seu uso. Esta ferramenta irá trocar dados com a planta simulada através do protocolo Modbus, onde os coils, mencionados anteriormente, carregam as informações do status de sensores e atuadores presentes na planta.

O controle supervisório desenvolvido com o auxílio da ferramenta DESTool será estruturado em um código na linguagem de programação C para, deste modo, ser implementado em um microcontrolador. A plataforma com microcontrolador embarcado chamada de Arduino foi escolhida para exercer esta função no presente projeto. A Figura 9 ilustra em um diagrama a estrutura básica do trabalho, explicitando as etapas em que sinais e eventos são empregados.

Ainda resta, no entanto, apresentar o modo utilizado para interpretar os sinais enviados pelas máquinas localizadas na planta simulada. É de interesse que estes sinais sejam conhecidos, uma vez que no código desenvolvido será necessário fazer referência aos coils corretos a fim de que o supervisório projetado funcione da maneira esperada. Com esta finalidade, portanto, foi analisado o arquivo com extensão .dev gerado pelo FlexFact quando o usuário deseja iniciar uma comunicação Modbus. Neste arquivo se encontram as informações que contém qual o número do coil (ou coils) que está (ou estão) associado(s) a um determinado evento (ou eventos). A Tabela 8 auxilia no entendimento do processo de interpretação citado.

Tabela 8 - conversão de sinais nos coils em eventos.

Fonte: Autor.

Coil	Máquina	Evento Controlável	Evento Não Controlável	Descrição
0	Conveyor Belt	✓	—	cb_boff $\begin{cases} coil\ 0 - false \\ coil\ 1 - false \end{cases}$
1	Conveyor Belt	✓	—	cb_bm+ $\begin{cases} coil\ 0 - true \\ coil\ 1 - false \end{cases}$ cb_bm- $\begin{cases} coil\ 0 - false \\ coil\ 1 - true \end{cases}$
2	Conveyor Belt	—	✓	cb_wpar cb_wplv
3	Stack Feeder	✓	—	sf_fdon sf_fdoff
4	Stack Feeder	—	✓	sf_fdhome
5	Stack Feeder	—	✓	sf_wpar sf_wplv
6	Exit Slide	—	✓	xs_wpar xs_wplv

Uma vez compreendida a estrutura base do presente trabalho, bem como a diferença entre sinais e eventos é possível, finalmente, partir para o entendimento do código elaborado para a implementação do sistema supervisório. A estrutura do sistema de controle supervisório utilizada no presente trabalho segue a proposta por (QUEIROZ e CURY, 2002) demonstrada na Figura 16.

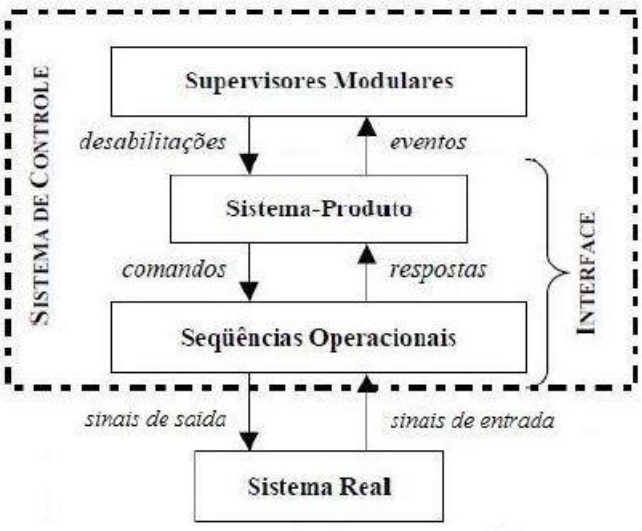

Figura 16 - estrutura do sistema de controle supervisório.

Fonte: QUEIROZ e CURY (2002).

Na Figura 16 o Sistema Real é a planta de manufatura simulada no software FlexFact do presente trabalho. Já a Interface, bem como os Supervisores Modulares (os quais são sintetizados segundo a Teoria de Controle Supervisório), está compreendida no microcontrolador embarcado na plataforma Arduino. A chamada Interface tem por função comandar o Sistema Real conforme os modelos abstratos das plantas e traduzir os comandos desses modelos em sinais de entrada do sistema real em respostas para os modelos abstratos. A abstração da planta, chamada de Sistema-Produto, envia comandos para executar as sequências operacionais referentes aos eventos não desabilitados e recebe respostas destas. Sempre que o sistema produto envia comandos ou recebe respostas, os supervisores são atualizados. As sequências operacionais correspondem ao conjunto de ações de baixo nível, tais como acessar o valor dos bits na mensagem Modbus a fim de modificar o valor dos coils que devem ser executadas, segundo a lógica demonstrada na Tabela 8, para realizar a ordem do comando enviado pelo Sistema-Produto.

Este conceito, idealizado na Figura 16, é que foi estruturado na linguagem de programação C e implementado no microcontrolador. A estrutura de código utilizada no presente trabalho foi fundamentada na proposta apresentada por (YURI et al., 2012) demonstrada na Figura 17. Esta proposta foi realizada para o controle de uma planta local, onde o sistema de controle possui aceso direto aos sinais, sensores e atuadores da planta. Portanto, esta proposta foi adaptada para que pudesse ser utilizada em um sistema onde planta e controle se comunicam através de um protocolo de comunicação, que é o caso do presente trabalho.

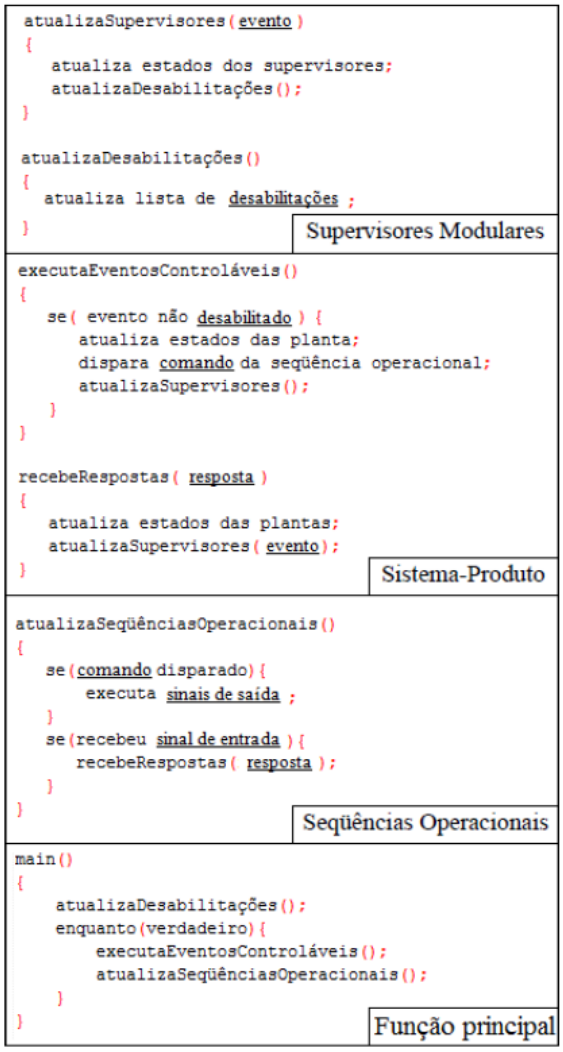

Figura 17 - estrutura do código na linguagem de programação C.

Fonte: YURI et al. (2012).

A função *atualizaSequênciasOperacionais()* constitui o nível mais baixo na hierarquia de controle. Sabe-se que as ações de controle do sistema supervisório são reações aos sinais enviados pelo Sistema Real, os quais devem ser constantemente acompanhados. A função *atualizaSequênciasOperacionais()* monitora a ocorrência destes sinais e responde ao sistema real com sinais de baixo nível de acordo com os comandos recebidos do Sistema-Produto. Quando ocorre um sinal de resposta do sistema real, a função *atualizaSequênciasOperacionais()* aciona o nível imediatamente superior – Sistema-Produto – a partir da função r*ecebeRespostas()*, a qual recebe como argumento a resposta representativa da ação ocorrida no sistema real. A função *executaEventosControláveis()*, juntamente com a função *recebeRespostas()*, constituem o nível do Sistema-Produto.

Segundo a teoria de controle supervisório, os eventos controláveis são executados quando não estiverem desabilitados. Para isso, o Sistema-Produto deve sempre verificar se há algum evento controlável que deva ser disparado, o que é feito pela função *executaEventosControláveis()* no corpo da função *main()*. Da maneira como foi implementado o código, quando há dois ou mais eventos não desabilitados, um deles é executado, a planta e o supervisório atualizam seus estados e, somente após, da mesma maneira, o evento subsequente é processado.

A função *recebeRespostas()* é responsável por tratar as respostas recebidas ao nível das sequências operacionais. Quando o Sistema-Produto dispara um evento controlável ou recebe sinalizações das sequências operacionais, os estados das plantas do Sistema-Produto são atualizados e o evento ocorrido é repassado ao nível do sistema supervisório através da função *atualizaSupervisores()*.

A função *atualizaSupervisores()* e *atualizaDesabilitações()* compõe o nível do sistema supervisório. Os estados dos supervisores são atualizados ao ser chamada a função *atualizaSupervisores()*, a qual atualiza a lista de eventos controláveis que devem ser desabilitados nos estados atuais através da função *atualizaDesabilitações()*. Quando ocorre uma sinalização do sistema real, somente devem ser atualizados os supervisores cujos estados sejam alterados por aquele evento.

Durante o desenvolvimento do trabalho, foi necessário o desenvolvimento de outras duas funções que não estão presentes na estrutura de código apresentada na Figura 17, são elas: *start_cmd()* e *select_sup_events()*. Esta necessidade foi sentida tendo em vista que o objetivo da estrutura desenvolvida é de sempre facilitar a utilização perante os usuários. Ou seja, a estrutura foi idealizada para ser confeccionada em blocos de tal maneira que máquinas podem ser adicionadas ou removidas da planta de manufatura sem causar grandes mudanças no código principal. Ainda, se grandes mudanças se fizerem necessárias, com esta utilização de uma sistemática em blocos, o usuário pode facilmente identificar quais funções devem ser alteradas sem prejudicar o funcionamento da lógica de programação.

A função *start_cmd()* pode ser interpretada como uma espécie de conversor entre sinais e eventos uma vez que nela ocorre a interpretação demonstrada na Tabela 8 desta vez, porém, na linguagem de programação C. A função *select_sup_events()*, por sua vez, tem por função apenas liberar os eventos controláveis para ocorrer tendo como condição o estado em que se encontra o modelo da máquina integrante da planta de manufatura em questão e do estado atual do sistema supervisório.

5 Resultados

O presente trabalho, diferentemente da maioria dos trabalhos de conclusão de curso no ramo da engenharia, não apresenta resultados na forma de gráficos ou figuras que devem ser interpretadas de maneira técnica a fim de gerar conclusões que qualifiquem a qualidade final do trabalho. Neste caso, o julgamento técnico aplicado tem por fundamento o método da comparação.

A situação que se tinha no estágio inicial do trabalho era apenas o estudo de caso simulado no FlexFact se comunicando com outro software (DESTool) para, desta maneira, impor o controle supervisório projetado sob a planta de manufatura virtual. Neste cenário, apesar do experimento ser baseado em uma planta de manufatura virtual, com boa aproximação da prática, o controle ainda é realizado na forma de autômatos sendo executados na ferramenta de software DESTool.

Na fase final do projeto, porém, o que se tinha era o Arduino, com o sistema supervisório já programado em linguagem C, se comunicando com a planta virtual a fim de exercer a mesma função que a estrutura inicial do trabalho exercia. Ou seja, a maneira mais correta de interpretar se o projeto proposto gerou resultados positivos é comparar o comportamento do sistema controlado pelo controle supervisório implementado no software DESTool com o comportamento apresentado pelo mesmo sistema quando submetido ao controle implementado no Arduino.

Desta maneira, através do método da comparação, foi observado que a planta de manufatura do estudo de caso se comportou da mesma maneira quando submetida aos dois métodos de implementação do controle supervisório. Ou seja, os resultados finais do projeto proposto foram satisfatórios tendo em vista que o “comportamento referência” da planta (quando o controle supervisório era simulado no software DESTool) foi muito similar ao comportamento visto quando o microcontrolador era o elemento que exercia o controle. Além disto, obteve-se um experimento que executa na prática o controle supervisório, aproximando o mesmo de um caso real.

6 Conclusões e Trabalhos Futuros

As conclusões sobre o projeto realizado foram otimistas, uma vez que o conceito inicial de estruturar um controle supervisório e implementá-lo em um microcontrolador de forma a exercer o controle sobre sistemas modelados a eventos discretos se provou factível e com resultados positivos.

Além disso, outro fator realizado com sucesso no presente trabalho foi a comunicação através do protocolo Modbus TCP entre a planta virtual e o Arduino. Este é um fator muito importante tendo em vista que este tipo de protocolo de comunicação é amplamente utilizado no ramo da indústria de manufatura, segmento do estudo de caso apresentado. Ou seja, os conceitos comprovados podem também ser facilmente adaptados e aplicados à plantas de manufatura físicas e não somente virtuais.

Ainda, pode-se afirmar que outro objetivo foi atingido: estruturar o código do controle supervisório em blocos. Isto se reflete em dizer que a proposta apresentada de trabalho pode ser aplicada em qualquer outra configuração de planta de manufatura, com mais ou menos máquinas, de uma maneira não muito custosa ao usuário, uma vez que o código foi estruturado em blocos separados e independentes.

Por fim, conclui-se que a proposta abordada neste trabalho de conclusão apresenta um ramo muito interessante de ser explorado podendo ser aplicada em virtualmente qualquer tipo de sistema capaz de ser modelado como um SED.

Durante a execução deste projeto foram observados alguns pontos que podem ser explorados mais afundo em trabalhos futuros, tais como:

- otimização do código em geral;

- estudo de caso com mais máquinas a fim de testar o conceito;

- investigação de ferramentas de geração automática de código a partir de autômatos;

- a eventualidade de dois ou mais eventos ocorrer juntos;

Sobre este último ponto ainda pode ser destacada a bibliografia de Fabian & Hellgren (1998) que aborda o tópico ocorrência de múltiplos eventos com maior nível de detalhe. Este ponto é ainda hoje motivo de investigação, pois advém da diferença entre sistemas síncronos (típicos em Controladores Lógicos Programáveis amplamente utilizados na indústria) e sistemas assíncronos (na qual se baseia a TCS).

7 Referências

ARDUINO. **Arduino Uno**. Disponível em: <http://arduino.cc/en/Main/ArduinoBoardUno>. Acesso em: 23 set. 2014.

_____. **Arduino Ethernet Shield**. Disponível em: <http://arduino.cc/en/Main/ArduinoEthernetShield>. Acesso em: 23 set. 2014.

EMBARCADOS. **Protocolo de Comunicação Modbus.** Disponível em: < http://www.embarcados.com.br/protocolo-modbus-fundamentos-e-aplicacoes/>. Acesso em: 23 set. 2014.

FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG. **DesTool.** Disponível em: < http://www.rt.eei.uni-erlangen.de/FGdes/destool/>. Acesso em: 23 set. 2014.

_____. **FlexFact**. Disponível em: <http://www.rt.eei.uni-erlangen.de/FGdes/flexfact.html>. Acesso em: 23 set. 2014.

MYARDUINOPROJECTS. **Biblioteca Mestre Modbus.** Disponível em: < http://myarduinoprojects.com/modbus.html>. Acesso em: 23 set. 2014.

MODBUSTOOLS. **Modbus Pool.** Disponível em: < http://www.modbustools.com/modbus_poll.asp>. Acesso em: 23 set. 2014.

_____. **Modbus Slave**. Disponível em: < http://www.modbustools.com/modbus_slave.asp>. Acesso em: 23 set. 2014.

MODBUS APPLICATION PROTOCOL SPECIFICATION. **Especificações do protocolo Modbus** Disponível em: < http://www.modbus.org/specs.php>. Acesso em: 23 set. 2014.

Queiroz, M. H. e Cury, J. E. R (2002). ; **Synthesis and implementation of local modular supervisory control for a manufacturing cell.** *Sixth International Workshop on Discrete Event Systems*.

Ramadge, P. J. G. e Wonham, W. M. (1989). **The control of discrete event systems.** *Proceedings of the IEEE,* 77**,** 81-98.

Yuri et al. (2010). **Síntese e implementação de controle supervisório modular local para um sistema de AGV.** XVIII Congresso Brasileiro de Automática.

Vieira, A. D. (2007) **Método de implementação do controle de sistemas a eventos discretos com aplicação da teoria de controle supervisório.**

Götz, M. (2013) **Apostila de Sistemas a Eventos Discretos, Autômatos & Teoria do Controle Supervisório**

Fabian, M. e Hellgren A. (1998) **PLC-based Implementation of Supervisory Control for Discrete Event Systems.** 37th IEEE Conference on Decision and Control, Tampa, Florida, USA, Dec 1998.

Anexos

Anexo 1: Biblioteca MgsModbus, anexo .h:

```
/*
  MgsModbus.h - an Arduino library for a Modbus TCP master and slave.
  V-0.1.1 Copyright (C) 2013  Marco Gerritse
  written and tested with Arduino 1.0
    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.
    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.
    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
  For this library the following library is used as start point:
    [1] Mudbus.h - an Arduino library for a Modbus TCP slave.
       Copyright (C) 2011  Dee Wykoff
    [2] Function codes 15 & 16 by Martin Pettersson
  The following references are used to write this library:
    [3] Open Modbus/Tcp specification, Release 1.0, 29 March 1999
       By Andy Swales, Schneider Electric
    [4] Modbus application protocol specification V1.1b3, 26 april 202
       From http:/www.modbus.org
  External software used for testing:
    [5] modpoll - www.modbusdriver.com/modpoll.html
    [6] ananas - www.tuomio.fi/ananas
    [7] mod_rssim - www.plcsimulator.org
    [8] modbus master - www.cableone.net/mblansett/
            This library use a single block of memory for all modbus data (mbData[] array). The  same data can
           be reached via several modbus functions, either via a 16 bit access or via an access bit. The length
           of MbData must at least 1.
             For the master the following modbus functions are implemented: 1, 2, 3, 4, 5, 6, 15, 16
              For the slave the following modbus functions are implemented: 1, 2, 3, 4, 5, 6, 15, 16
  The internal and external addresses are 0 (zero) based
  V-0.1.1 2013-06-02
  bugfix
  V-0.1.0 2013-03-02
  initinal version
*/
#include "Arduino.h"
#include <SPI.h>
#include <Ethernet.h>
#ifndef MgsModbus_h
#define MgsModbus_h
#define MbDataLen 30 // length of the MdData array
#define MB_PORT 1502
enum MB_FC {
  MB_FC_NONE  = 0,
  MB_FC_READ_COILS    = 1,
  MB_FC_READ_DISCRETE_INPUT  = 2,
  MB_FC_READ_REGISTERS   = 3,
  MB_FC_READ_INPUT_REGISTER   = 4,
  MB_FC_WRITE_COIL  = 5,
```

```
  MB_FC_WRITE_REGISTER = 6,
  MB_FC_WRITE_MULTIPLE_COILS  = 15,
  MB_FC_WRITE_MULTIPLE_REGISTERS = 16
            };
class MgsModbus
{
public:
  // general
  MgsModbus();
              word MbData[MbDataLen]; // memory block that holds all the modbus user data
  boolean GetBit(word Number);
              boolean SetBit(word Number,boolean Data); // returns true when the number is in the MbData
             range
  // modbus master
  void Req(MB_FC FC, word Ref, word Count, word Pos);
  void MbmRun();
  IPAddress remSlaveIP;
  // modbus slave
  void MbsRun();
              word GetDataLen();
  boolean Check_return(); // Check if slave returned message
private:
  // general
  MB_FC SetFC(int fc);
  // modbus master
  uint8_t MbmByteArray[260]; // send and recieve buffer
  MB_FC MbmFC;
  int MbmCounter;
  void MbmProcess();
  word MbmPos;
  word MbmBitCount;
  //modbus slave
  uint8_t MbsByteArray[260]; // send and recieve buffer
  MB_FC MbsFC;
  MB_FC MbsFC_wait; //to synchronize send/receive messages
};
#endif
```

Anexo 2: Biblioteca MgsModbus, anexo .cpp:

```
#include "MgsModbus.h"
// For Arduino 1.0
EthernetServer MbServer(MB_PORT);
EthernetClient MbmClient;
 //#define DEBUG
 #define DEBUG2
 int retorno;
MgsModbus::MgsModbus()
{
}
//****************** Send data for ModBusMaster ***************
void MgsModbus::Req(MB_FC FC, word Ref, word Count, word Pos)
{
  MbmFC = FC;
  MbsFC_wait = FC;
  byte ServerIp[] = {255,255,255,255};//Cria variável inicializada com lixo
              // Copia em ServerIp(tipo byte[]) o endereço de remSlaveIP(tipo IPAddress) setado pelo usuário
  ServerIp[0] = remSlaveIP[0];
  ServerIp[1] = remSlaveIP[1];
```

```
ServerIp[2] = remSlaveIP[2];
ServerIp[3] = remSlaveIP[3];

MbmByteArray[0] = 0;  // ID high byte
MbmByteArray[1] = 1;  // ID low byte
MbmByteArray[2] = 0;  // protocol high byte
MbmByteArray[3] = 0;  // protocol low byte
MbmByteArray[5] = 6;  // Length low byte;
MbmByteArray[4] = 0;  // Length high byte
MbmByteArray[6] = 1;  // unit ID
MbmByteArray[7] = FC; // function code
MbmByteArray[8] = highByte(Ref);
MbmByteArray[9] = lowByte(Ref);
          //****************** Read Coils (1) & Read discrete Inputs (2) **********************
if(FC == MB_FC_READ_COILS || FC == MB_FC_READ_DISCRETE_INPUT) {
  if (Count < 1) {Count = 1;}
  if (Count > 125) {Count = 2000;}
  MbmByteArray[10] = highByte(Count);
  MbmByteArray[11] = lowByte(Count);
}
          //****************** Read Registers (3) & Read Input registers (4) ******************
if(FC == MB_FC_READ_REGISTERS || FC == MB_FC_READ_INPUT_REGISTER) {
  if (Count < 1) {Count = 1;}
  if (Count > 125) {Count = 125;}
  MbmByteArray[10] = highByte(Count);
  MbmByteArray[11] = lowByte(Count);
}
//****************** Write Coil (5) **********************
if(MbmFC == MB_FC_WRITE_COIL) {
            if (GetBit(Pos)) {MbmByteArray[10] = 0xFF;} else {MbmByteArray[10] = 0;} //   0xFF coil on 0x00
          coil off
  MbmByteArray[11] = 0; // always zero
}
//****************** Write Register (6) ******************
if(MbmFC == MB_FC_WRITE_REGISTER) {
  MbmByteArray[10] = highByte(MbData[Pos]);
  MbmByteArray[11] = lowByte(MbData[Pos]);
}
//****************** Write Multiple Coils (15) **********************
// not fully tested
if(MbmFC == MB_FC_WRITE_MULTIPLE_COILS) {
  if (Count < 1) {Count = 1;}
  if (Count > 800) {Count = 800;}
  MbmByteArray[10] = highByte(Count);
  MbmByteArray[11] = lowByte(Count);
  MbmByteArray[12] = (Count + 7) /8;
  MbmByteArray[4] = highByte(MbmByteArray[12] + 7); // Length high byte
  MbmByteArray[5] = lowByte(MbmByteArray[12] + 7); // Length low byte;
  for (int i=0; i<Count; i++) {
    bitWrite(MbmByteArray[13+(i/8)],i-((i/8)*8),GetBit(Pos+i));
  }
}
//****************** Write Multiple Registers (16) ******************
if(MbmFC == MB_FC_WRITE_MULTIPLE_REGISTERS) {
  if (Count < 1) {Count = 1;}
  if (Count > 100) {Count = 100;}
```

```
  MbmByteArray[10] = highByte(Count);
  MbmByteArray[11] = lowByte(Count);
  MbmByteArray[12] = (Count*2);
  MbmByteArray[4] = highByte(MbmByteArray[12] + 7); // Length high byte
  MbmByteArray[5] = lowByte(MbmByteArray[12] + 7); // Length low byte;
  for (int i=0; i<Count;i++) {
   MbmByteArray[(i*2)+13] = highByte (MbData[Pos + i]);
   MbmByteArray[(i*2)+14] = lowByte (MbData[Pos + i]);
  }
 }
 //****************** ?? ******************
 //Serial.print("ServerIp= ");
 //Serial.println(ServerIp);
 /* Serial.println(ServerIp[0]);
 Serial.println(ServerIp[1]);
 Serial.println(ServerIp[2]);
 Serial.println(ServerIp[3]); */
 retorno=MbmClient.connect(ServerIp,MB_PORT);
 if (retorno == 1) {
  #ifdef DEBUG
   Serial.println("connected with modbus slave");
   Serial.print("Master request: ");
   for(int i=0;i<MbmByteArray[5]+6;i++) {
    if(MbmByteArray[i] < 16){Serial.print("0");}
    Serial.print(MbmByteArray[i],HEX);
    if (i != MbmByteArray[5]+5) {Serial.print(".");} else {Serial.println();}
   }
  #endif
  //for(int i=0;i<MbmByteArray[5]+6;i++) {
  //  MbmClient.write(MbmByteArray[i]);
  //}
        MbmClient.write(MbmByteArray,MbmByteArray[5]+6); //Envia mensagem de uma só vez ao invés
        de byte a byte
  MbmCounter = 0;
  MbmByteArray[7] = 0;
  MbmPos = Pos;
  MbmBitCount = Count;
 } else {
  #ifdef DEBUG2
   Serial.println("connection with modbus master failed");
     Serial.println(retorno);
  #endif
  MbmClient.stop();
 }
}
//****************** Receive data for ModBusMaster ****************
void MgsModbus::MbmRun()
{
    if (MbmClient.connected()){
    //Serial.println("Connected yes");
    }
 //****************** Read from socket ****************
 while (MbmClient.available()) {
  MbmByteArray[MbmCounter] = MbmClient.read();
    //Serial.println(MbmByteArray[MbmCounter]);
  if (MbmCounter > 4)  {
   if (MbmCounter == MbmByteArray[5] + 5) { // the full answer is received
    MbmClient.stop();
    MbmProcess();
    #ifdef DEBUG
```

```
      Serial.println("receive clear");
     #endif
    }
   }
   MbmCounter++;
  }
}

void MgsModbus::MbmProcess()
{
  MbsFC_wait = MB_FC_NONE;
  MbmFC = SetFC(int (MbmByteArray[7]));
  #ifdef DEBUG
   for (int i=0;i<MbmByteArray[5]+6;i++) {
    if(MbmByteArray[i] < 16) {Serial.print("0");}
    Serial.print(MbmByteArray[i],HEX);
    if (i != MbmByteArray[5]+5) {Serial.print(".");
    } else {Serial.println();}
   }
  #endif
            //****************** Read Coils (1) & Read discrete Inputs (2) **********************
            if(MbmFC == MB_FC_READ_COILS || MbmFC == MB_FC_READ_DISCRETE_INPUT) {
   word Count = MbmByteArray[8] * 8;
   if (MbmBitCount < Count) {
    Count = MbmBitCount;
   }
   for (int i=0;i<Count;i++) {
    if (i + MbmPos < MbDataLen * 16) {
     SetBit(i + MbmPos,bitRead(MbmByteArray[(i/8)+9],i-((i/8)*8)));
    }
   }
  }
            //****************** Read Registers (3) & Read Input registers (4) ******************
            if(MbmFC == MB_FC_READ_REGISTERS || MbmFC == MB_FC_READ_INPUT_REGISTER) {
   word Pos = MbmPos;
   for (int i=0;i<MbmByteArray[8];i=i+2) {
    if (Pos < MbDataLen) {
     MbData[Pos] = (MbmByteArray[i+9] * 0x100) + MbmByteArray[i+1+9];
     Pos++;
    }
   }
  }
  //****************** Write Coil (5) **********************
  if(MbmFC == MB_FC_WRITE_COIL){
  }
  //****************** Write Register (6) ******************
  if(MbmFC == MB_FC_WRITE_REGISTER){
  }
  //****************** Write Multiple Coils (15) **********************
  if(MbmFC == MB_FC_WRITE_MULTIPLE_COILS){
  }
  //****************** Write Multiple Registers (16) ******************
  if(MbmFC == MB_FC_WRITE_MULTIPLE_REGISTERS){
  }
}
```

```
//****************** Receive data for ModBusSlave ****************
void MgsModbus::MbsRun()
{
 //****************** Read from socket ****************
 EthernetClient client = MbServer.available();
 if(client.available())
 {
  delay(10);
  int i = 0;
  while(client.available())
  {
   MbsByteArray[i] = client.read();
   i++;
  }
  MbsFC = SetFC(MbsByteArray[7]);  //Byte 7 of request is FC
 }
 int Start, WordDataLength, ByteDataLength, CoilDataLength, MessageLength;
 //****************** Read Coils (1 & 2) **********************
            if(MbsFC == MB_FC_READ_COILS || MbsFC == MB_FC_READ_DISCRETE_INPUT) {
  Start = word(MbsByteArray[8],MbsByteArray[9]);
  CoilDataLength = word(MbsByteArray[10],MbsByteArray[11]);
  ByteDataLength = CoilDataLength / 8;
  if(ByteDataLength * 8 < CoilDataLength) ByteDataLength++;
  CoilDataLength = ByteDataLength * 8;
  MbsByteArray[5] = ByteDataLength + 3; //Number of bytes after this one.
             MbsByteArray[8] = ByteDataLength;     //Number of bytes after this one (or number of bytes of
          data).
  for(int i = 0; i < ByteDataLength ; i++)
  {
   MbsByteArray[9 + i] = 0; // To get all remaining not written bits zero
   for(int j = 0; j < 8; j++)
   {
    bitWrite(MbsByteArray[9 + i], j, GetBit(Start + i * 8 + j));
   }
  }
  MessageLength = ByteDataLength + 9;
  client.write(MbsByteArray, MessageLength);
  MbsFC = MB_FC_NONE;
 }
 //****************** Read Registers (3 & 4) ******************
            if(MbsFC == MB_FC_READ_REGISTERS || MbsFC == MB_FC_READ_INPUT_REGISTER) {
  Start = word(MbsByteArray[8],MbsByteArray[9]);
  WordDataLength = word(MbsByteArray[10],MbsByteArray[11]);
  ByteDataLength = WordDataLength * 2;
  MbsByteArray[5] = ByteDataLength + 3; //Number of bytes after this one.
             MbsByteArray[8] = ByteDataLength;     //Number of bytes after this one (or number of bytes of
          data).
  for(int i = 0; i < WordDataLength; i++)
  {
   MbsByteArray[ 9 + i * 2] = highByte(MbData[Start + i]);
   MbsByteArray[10 + i * 2] =  lowByte(MbData[Start + i]);
  }
  MessageLength = ByteDataLength + 9;
  client.write(MbsByteArray, MessageLength);
  MbsFC = MB_FC_NONE;
 }
 //****************** Write Coil (5) **********************
 if(MbsFC == MB_FC_WRITE_COIL) {
  Start = word(MbsByteArray[8],MbsByteArray[9]);
```

```
  if (word(MbsByteArray[10],MbsByteArray[11]) == 0xFF00){SetBit(Start,true);}
            if (word(MbsByteArray[10],MbsByteArray[11]) == 0x0000){SetBit(Start,false);}
  MbsByteArray[5] = 2; //Number of bytes after this one.
  MessageLength = 8;
  client.write(MbsByteArray, MessageLength);
  MbsFC = MB_FC_NONE;
 }
 //****************** Write Register (6) ******************
 if(MbsFC == MB_FC_WRITE_REGISTER) {
  Start = word(MbsByteArray[8],MbsByteArray[9]);
  MbData[Start] = word(MbsByteArray[10],MbsByteArray[11]);
  MbsByteArray[5] = 6; //Number of bytes after this one.
  MessageLength = 12;
  client.write(MbsByteArray, MessageLength);
  MbsFC = MB_FC_NONE;
 }
 //****************** Write Multiple Coils (15) **********************
 if(MbsFC == MB_FC_WRITE_MULTIPLE_COILS) {
  Start = word(MbsByteArray[8],MbsByteArray[9]);
  CoilDataLength = word(MbsByteArray[10],MbsByteArray[11]);
  MbsByteArray[5] = 6;
  for(int i = 0; i < CoilDataLength; i++)
  {
   SetBit(Start + i,bitRead(MbsByteArray[13 + (i/8)],i-((i/8)*8)));
  }
  MessageLength = 12;
  client.write(MbsByteArray, MessageLength);
  MbsFC = MB_FC_NONE;
 }
 //****************** Write Multiple Registers (16) ******************
 if(MbsFC == MB_FC_WRITE_MULTIPLE_REGISTERS) {
  Start = word(MbsByteArray[8],MbsByteArray[9]);
  WordDataLength = word(MbsByteArray[10],MbsByteArray[11]);
  ByteDataLength = WordDataLength * 2;
  MbsByteArray[5] = 6;
  for(int i = 0; i < WordDataLength; i++)
  {
            MbData[Start + i] =  word(MbsByteArray[ 13 + i * 2],MbsByteArray[14 + i * 2]);
  }
  MessageLength = 12;
  client.write(MbsByteArray, MessageLength);
  MbsFC = MB_FC_NONE;
 }
}


//****************** ?? ******************
MB_FC MgsModbus::SetFC(int fc)
{
 MB_FC FC;
 FC = MB_FC_NONE;
 if(fc == 1) FC = MB_FC_READ_COILS;
 if(fc == 2) FC = MB_FC_READ_DISCRETE_INPUT;
 if(fc == 3) FC = MB_FC_READ_REGISTERS;
 if(fc == 4) FC = MB_FC_READ_INPUT_REGISTER;
 if(fc == 5) FC = MB_FC_WRITE_COIL;
```

```
 if(fc == 6) FC = MB_FC_WRITE_REGISTER;
 if(fc == 15) FC = MB_FC_WRITE_MULTIPLE_COILS;
 if(fc == 16) FC = MB_FC_WRITE_MULTIPLE_REGISTERS;
 return FC;
}


word MgsModbus::GetDataLen()
{
 return MbDataLen;
}


boolean MgsModbus::GetBit(word Number)
{
 int ArrayPos = Number / 16;
 int BitPos = Number - ArrayPos * 16;
 boolean Tmp = bitRead(MbData[ArrayPos],BitPos);
 return Tmp;
}

boolean MgsModbus::Check_return()
{
 boolean Tmp = true;
 if (MbsFC_wait!=MB_FC_NONE)
     Tmp = false;
 return Tmp;
}



boolean MgsModbus::SetBit(word Number,boolean Data)
{
 int ArrayPos = Number / 16;
 int BitPos = Number - ArrayPos * 16;
 boolean Overrun = ArrayPos > MbDataLen * 16; // check for data overrun
 if (!Overrun){
  bitWrite(MbData[ArrayPos],BitPos,Data);
 }
 return Overrun;
}
```

Apêndices

Apêndice 1: função main() do código:

```
#include <SPI.h>

#include <Ethernet.h>

#include "MgsModbus.h"

//#define TEST

MgsModbus Mb;

int inByte = 0; // incoming serial byte

IPAddress ip(192,168,1,177); // endereço do Arduino -> Cliente

IPAddress SlaveAddress(192,168,1,155); // endereço do PC -> Server

int state_sf = 1; //first state of the stack feeder machine

int state_cb = 1; //first state of the conveyor belt machine

int state_xs = 1; //first state of the exit slide machine

int state_sup = 1; //first state of the supervisor finite state machine

//inicializa eventos controláveis

int sf1_fdon = 0;

int sf1_fdoff = 0;

int cb1_bmeast = 0;

int cb1_boff = 0;

//inicializa eventos não controláveis

int sf1_fdhome = 0;

int sf1_wpar = 0;

int sf1_wplv = 0;

int xs1_wpar = 0;

int xs1_wplv = 0;

//inicializa variáveis do supervisório (desabilitações)

int sup_sf1_fdon = 0;

int sup_sf1_fdoff = 0;
```

```
int sup_cb1_bmeast = 0;

int sup_cb1_boff = 0;

//inicializa variáveis utilizadas em recebeRespostas

int MbData_passado[] = {0, 0, 0, 0, 0, 0, 0};

int MbData_atual[] = {1, 1, 1, 1, 1, 1, 1};

int MbData_comp[] = {2, 2, 2, 2, 2, 2, 2};

int sensor_change; //se precisar

void setup()

{

 // serial setup

 Serial.begin(9600);

 // ethernet via dhcp

 uint8_t mac[]    = { 0x90, 0xA2, 0xDA, 0x0F, 0x98, 0xC0 }; //mac of the Ethernet Shield

 Ethernet.begin(mac,ip);

 Serial.print("Client is at ");

 Serial.println(Ethernet.localIP());

 Mb.remSlaveIP = SlaveAddress; // endereço do PC -> Server

 #ifdef TEST

  Serial.print("estado stack feeder:");

  Serial.println(state_sf);

  Serial.print("estado conveyor belt:");

  Serial.println(state_cb);

  Serial.print("estado exit slide:");

  Serial.println(state_xs);

  Serial.print("estado supervisorio:");

  Serial.println(state_sup);

  Serial.println("Status dos eventos controlaveis:(1 habilitado/0 desabilitado)");

  Serial.print("sf1_fdon:");

  Serial.println(sup_sf1_fdon);

  Serial.print("sf1_fdoff:");
```

```
    Serial.println(sup_sf1_fdoff);
    Serial.print("cb1_bm+:");
    Serial.println(sup_cb1_bmeast);
    Serial.print("cb1_boff:");
    Serial.println(sup_cb1_boff);
    Serial.println("Pressione a tecla 0 para mudanca de estado");
    Serial.println("Pressione a tecla 1 para observar o estado atual");
    Serial.println("Pressione a tecla 2 para observar o valor atual dos coils");
  #endif
  atualizaDesabilitacoes();
  ReadCoils();
}
void loop()
{
   if (Serial.available() > 0)
  {
#ifdef TEST
  //// get incoming byte:////
    inByte = Serial.read();
    if (inByte == '1')
    {
      Serial.println();
      Serial.print("estado stack feeder:");
      Serial.println(state_sf);
      Serial.print("estado conveyor belt:");
      Serial.println(state_cb);
      Serial.print("estado exit slide:");
      Serial.println(state_xs);
      Serial.print("estado supervisorio:");
```

```
  Serial.println(state_sup);
  Serial.println("Status dos eventos controlaveis:(1 habilitado/0 desabilitado)");
  Serial.print("sf1_fdon:");
  Serial.println(sup_sf1_fdon);
  Serial.print("sf1_fdoff:");
  Serial.println(sup_sf1_fdoff);
  Serial.print("cb1_bm+:");
  Serial.println(sup_cb1_bmeast);
  Serial.print("cb1_boff:");
  Serial.println(sup_cb1_boff);
  Serial.println();
  Serial.println("Pressione a tecla 0 para mudanca de estado");
  Serial.println("Pressione a tecla 1 para observar o estado atual");
  Serial.println("Pressione a tecla 2 para observar o valor atual dos coils");
 }
 if (inByte == '2')
 {
  Serial.println("MbData:");
  for (int j=0;j<7;j++)
    {
    Serial.print("Coil ");
    Serial.print(j);
    Serial.print(" :");
    Serial.println(Mb.GetBit(j));
    }
  Serial.println("Pressione a tecla 0 para mudanca de estado");
  Serial.println("Pressione a tecla 1 para observar o estado atual");
  Serial.println("Pressione a tecla 2 para observar o valor atual dos coils");
 }
 if (inByte == '0')
```

```
  {


#endif
  executaEventosControlaveis();
  atualizaSequenciasOperacionais();
#ifdef TEST
  Serial.print("estado stack feeder:");
  Serial.println(state_sf);
  Serial.print("estado conveyor belt:");
  Serial.println(state_cb);
  Serial.print("estado exit slide:");
  Serial.println(state_xs);
  Serial.print("estado supervisorio:");
  Serial.println(state_sup);
  Serial.println("Status dos eventos controlaveis:(1 habilitado/0 desabilitado)");
  Serial.print("sf1_fdon:");
  Serial.println(sup_sf1_fdon);
  Serial.print("sf1_fdoff:");
  Serial.println(sup_sf1_fdoff);
  Serial.print("cb1_bm+:");
  Serial.println(sup_cb1_bmeast);
  Serial.print("cb1_boff:");
  Serial.println(sup_cb1_boff);
  Serial.println("Pressione a tecla 0 para mudanca de estado");
  Serial.println("Pressione a tecla 1 para observar o estado atual");
  Serial.println("Pressione a tecla 2 para observar o valor atual dos coils");
  }
  delay(1000);
#endif
```

```
 }
}
```

Apêndice 2: função Conveyor_belt() do código:

```
void Conveyor_belt()
{
#ifdef TEST
 Serial.println("CB");
#endif
 switch(state_cb)
 {
   case 1:
    if(cb1_bmeast == 1) //conveyor belt motor on to the east
     state_cb = 2;    //cb1_bm+
    break;
   case 2:
    if(cb1_boff == 1)  //conveyor belt motor off
     state_cb = 1;   //cb1_boff
    break;
 }
}
```

Apêndice 3: função Exit_slide() do código:

```
void Exit_slide()
{
#ifdef TEST
 Serial.println("XS");
#endif
 switch(state_xs)
 {
   case 1:
    if(xs1_wpar == 1) //wait for the workpiece sensor positive edge arrival
```

```
      state_xs = 2;   //xs1_wpar
     break;
   case 2:
    if(xs1_wplv == 1) //wait for the workpiece sensor negative edge arrival
      state_xs = 1;      //xs1_wplv
    break;
 }
}
```

Apêndice 4: função Exit_slide() do código:

```
void Exit_slide()
{
#ifdef TEST
 Serial.println("XS");
#endif
 switch(state_xs)
 {
  case 1:
    if(xs1_wpar == 1) //wait for the workpiece sensor positive edge arrival
      state_xs = 2;   //xs1_wpar
    break;
  case 2:
    if(xs1_wplv == 1) //wait for the workpiece sensor negative edge arrival
      state_xs = 1;      //xs1_wplv
    break;
 }
}
```

Apêndice 5: função ReadCoils() do código:

```
void ReadCoils()
{
```

```
#ifdef TEST
 Serial.println("Read Coils");
#endif
 Mb.Req(MB_FC_READ_COILS,0,7,0);// 1 // ref, count, pos
                //ref: endereço base no escravo
                //count: numero de bits a ser lido do escravo
                //pos: 0x10 hex = 16 decimal -> se refere a posição do bit em
                //MbData onde serão escritos os bits lidos no escravo
 while(Mb.Check_return() == false)
 {
  Mb.MbmRun();
 }
}
```

Apêndice 6: função Stack_feeder() do código:

```
void Stack_feeder()
{
#ifdef TEST
 Serial.println("SF");
#endif
 switch(state_sf)
 {
  case 1:
   if(sf1_wpar == 1) //wait for the workpiece sensor positive edge arrival
   {
    state_sf = 2;   //sf1_wpar
    sf1_wpar = 0;   //evento não presente no supervisor deve ser resetado aqui
   }
   break;
  case 2:
   if(sf1_fdon == 1) //feed motor on
```

```
      state_sf = 3;   //sf1_fdon
     break;
   case 3:
    if(sf1_fdhome == 1) //wait for the pusher sensor positive edge arrival
    {
      state_sf = 4;     //sf1_fdhome
      sf1_fdhome = 0;  // evento não presente no supervisor deve ser resetado aqui
    }
    break;
   case 4:
    if(sf1_fdoff == 1) //feed motor off
      state_sf = 1;    //sf1_fdoff
    break;
 }
}
```

Apêndice 7: função WriteCoils() do código:

```
void WriteCoils()
{
#ifdef TEST
 Serial.println("Write Coils");
#endif
   Mb.Req(MB_FC_WRITE_MULTIPLE_COILS,     0,7,0);
   while(Mb.Check_return() == false)
   {
    Mb.MbmRun();
   }
}
```

Apêndice 8: função atualizaDesabilitacoes() do código:

```
void atualizaDesabilitacoes()
```

```
{
#ifdef TEST
 Serial.println("atualizaDesabilitacoes");
 //Serial.println(state_sup);
#endif
 //desabilita inicialmente os eventos controláveis do supervisório
 sup_sf1_fdon = 0;
 sup_sf1_fdoff = 0;
 sup_cb1_bmeast = 0;
 sup_cb1_boff = 0;
 //rotina que habilita os eventos controláveis baseado no estado atual do supervisor
 switch(state_sup)
 {
  case 1:
   sup_sf1_fdon = 0;
   sup_sf1_fdoff = 1;
   sup_cb1_bmeast = 1;
   sup_cb1_boff = 1;
   break;
  case 2:
   sup_sf1_fdon = 1;
   sup_sf1_fdoff = 1;
   sup_cb1_bmeast = 1;
   sup_cb1_boff = 1;
   break;
  case 3:
   sup_sf1_fdon = 1;
   sup_sf1_fdoff = 1;
   sup_cb1_bmeast = 0;
   sup_cb1_boff = 0;
```

```
    break;
   case 4:
    sup_sf1_fdon = 0;
    sup_sf1_fdoff = 1;
    sup_cb1_bmeast = 0;
    sup_cb1_boff = 0;
    break;
 }
}
```

Apêndice 9: função atualizaSequenciasOperacionais() do código:

```
void atualizaSequenciasOperacionais()
{
#ifdef TEST
 Serial.println("atualizaSequenciasOperacionais");
#endif
 WriteCoils();  //escreve nos coils os sinais modificados em start_cmd
 for(int k=0;k<7;k++) //coleta os valore de MbData no instante após em que os
 {              //valores dos coils foram modificados em start_cmd
  MbData_passado[k] = Mb.GetBit(k);
 }
 ReadCoils();  //coleta o valor dos coils em MbData
 recebeRespostas();
}
```

Apêndice 10: função atualizaSupervisores() do código:

```
void atualizaSupervisores()
{
 switch(state_sup)
 {
  case 1:
```

```
  if(cb1_bmeast == 1 || cb1_boff == 1)
  {
   state_sup = 3;
   cb1_bmeast = 0;
   cb1_boff = 0;
  }
  if(sf1_fdoff == 1)
  {
   state_sup = 2;
   sf1_fdoff = 0;
  }
  if(xs1_wpar == 1 || xs1_wplv == 1)
  {
   state_sup = 4;
   xs1_wpar = 0;
   xs1_wplv = 0;
  }
  break;
 case 2:
  if(sf1_fdon == 1 || sf1_fdoff == 1)
  {
   state_sup = 1;
   sf1_fdon = 0;
   sf1_fdoff = 0;
  }
  if(cb1_boff == 1 || cb1_bmeast == 1)
  {
   state_sup = 4;
   cb1_boff = 0;
   cb1_bmeast = 0;
```

```
  }
  if(xs1_wpar == 1 || xs1_wplv == 1)
  {
    state_sup = 3;
    xs1_wpar = 0;
    xs1_wplv = 0;
  }
  break;
 case 3:
  if(sf1_fdon == 1 || sf1_fdoff == 1)
  {
    state_sup = 4;
    sf1_fdon = 0;
    sf1_fdoff = 0;
  }
  if(xs1_wpar == 1 || xs1_wplv == 1)
  {
    state_sup = 2;
    xs1_wpar = 0;
    xs1_wplv = 0;
  }
  break;
 case 4:
  if(sf1_fdoff == 1)
  {
    state_sup = 3;
    sf1_fdoff = 0;
  }
  if(xs1_wpar == 1 || xs1_wplv == 1)
```

```
    {
      state_sup = 1;
      xs1_wpar = 0;
      xs1_wplv = 0;
    }
    break;
  }
  atualizaDesabilitacoes();
}
```

Apêndice 11: função executaEventosControlaveis() do código:

```
void executaEventosControlaveis()
{
#ifdef TEST
  Serial.println("executaEventosControlaveis");
#endif
  select_cont_events();
  atualizaSupervisores();
}
```

Apêndice 12: função recebeRespostas() do código:

```
void recebeRespostas()
{
#ifdef TEST
  Serial.println("recebeRespostas");
#endif
  for(int k=0;k<7;k++) //coleta os valore de MbData no instante após em que os
  {              //valores dos coils foram lidos atualizaSequenciasOperacionais
    MbData_atual[k] = Mb.GetBit(k);
  }
  //compara o valor dos coils deste instante com os valores destes
  //no instante anterior para identificar se algum evento não
```

```
 //controlável (sensor) ocorreu.
 for(int k=0;k<7;k++)
 {
  if(MbData_atual[k] == MbData_passado[k]) //não houve mudança nos sensores
    MbData_comp[k] = 0;
  if(MbData_atual[k] > MbData_passado[k]) //detecta a chegada de uma borda positiva no sensor
  {
    MbData_comp[k] = 1;
    //sensor_change++;
  }
  if(MbData_atual[k] < MbData_passado[k]) //detecta a chegada de uma borda negativa no sensor
  {
    MbData_comp[k] = -1;
    //sensor_change++;
  }
 }
 start_cmd;
 atualizaSupervisores();
}
```

Apêndice 13: função select_cont_events() do código:

```
void select_cont_events()
{
 if(state_cb == 1 && sup_cb1_bmeast == 1)
   cb1_bmeast = 1; //evento cb1_bmeast está liberado para ser executado
 if(state_cb == 2 && sup_cb1_boff == 1)
   cb1_boff = 1;
 if(state_sf == 2 && sup_sf1_fdon == 1)
   sf1_fdon = 1;
 if(state_sf == 4 && sup_sf1_fdoff == 1)
```

```
   sf1_fdoff = 1;
  start_cmd();
}
```

Apêndice 14: função start_cmd() do código:

```
void start_cmd()
{
 ////////   EVENTOS CONTROLÁVEIS   ////////
 if(cb1_bmeast == 1)
 {
   Mb.SetBit(0,true);
   Mb.SetBit(1,false);
 }
 if(cb1_boff == 1)
 {
   Mb.SetBit(0,false);
   Mb.SetBit(1,false);
 }
 if(sf1_fdon == 1)
   Mb.SetBit(3,true);
 if(sf1_fdoff == 1)
   Mb.SetBit(3,false);
 ////////   EVENTOS NÃO CONTROLÁVEIS   ////////
 //  if(MbData_comp[2] == 1)
 //    cb1_wpar = 1;
 //  if(MbData_comp[2] == -1)
 //    cb1_wplv = 1;
 if(MbData_comp[4] == 1)
   sf1_fdhome = 1;
 if(MbData_comp[5] == 1)
   sf1_wpar = 1;
```

```
  if(MbData_comp[5] == -1)
    sf1_wplv = 1;
  if(MbData_comp[6] == 1)
    xs1_wpar = 1;
  if(MbData_comp[6] == -1)
    xs1_wplv = 1;
  Stack_feeder();
  Conveyor_belt();
  Exit_slide();
 }
```