Faculdade de Engenharia da Universidade do Porto

Controlo de uma Persiana

Carlos Pinto
Sara Noronha

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores
Sistemas Baseados em Microprocessadores

Dezembro de 2018

Resumo

Este documento descreve todo o processo necessário para o controlo de uma persiana através de um circuito baseado num microcontrolador (ATMega328p inserido numa placa Arduino Uno), tanto a nível de hardware, como de software. A componente de hardware não será tão abordada pois o circuito de interface com a persiana já nos era fornecido, não havendo necessidade do seu redimensionamento.

Abstract

This document describes the whole process of designing, programming and assembling a circuit based on a microcontroller (ATMega328p embedded on a Arduino Uno board) that controls a shutter. Each chapter describes these points concerning each phase of the project, including hardware e software. The hardware phase won't be extensively approached as the interface circuit was already supplied to us from the beginning, not being necessary to size it.

Índice

Resumo 2
Abstract 4
Capítulo 1 1
Introdução 1
Capítulo 2 3
Hardware 3
2.1- Circuito Interface com a persiana 3
2.2 Botões 4
Capítulo 3 5
Software 5
3.1 - Máquina de Estados 5
3.2 – Código 6
3.2.1 – Inicializações 8
3.2.2- Interrupções 10
3.2.3 – Switch – máquina de estados 11
Conclusão 12
Anexos 13
Esquema do circuito completo 13
Código final (main.c) 14
Código final (biblioteca “serial” – header file) 25
Código final (biblioteca “serial” – library file) 26
Referências 27

Capítulo 1
Introdução

Este trabalho foi realizado com o objetivo de controlar uma persiana através de um circuito que nos foi dado desde início e apresentado no próximo capítulo. Após o estudo do circuito, fizemos testes para ver como o circuito se comportava quando alimentado. Depois de entendermos como o circuito se comportava quando alimentado, passámos para o software, em que desenvolvemos código capaz de controlar esta persiana e que vai ser apresentado e explicado mais à frente.

Capítulo 2
Hardware

2.1- Circuito Interface com a persiana

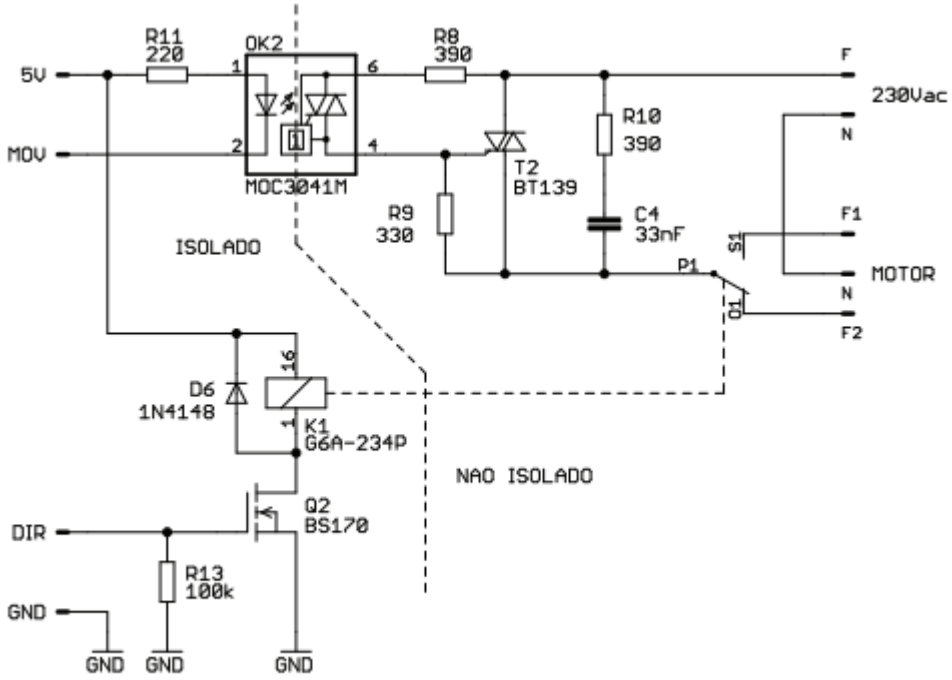


Figura 1 – Circuito de interface com a persiana.

Neste circuito temos quatro sinais – GND, 5V, MOV, DIR que devem ser ligados entre a placa de interface e a placa de controlo.
O sinal MOV é ativo a zero e permite colocar a persiana em movimento gerando a tensão necessária para que o triac T2 conduza corrente, servindo este como interruptor de corrente alternada. A persiana tem também um mecanismo de segurança que para sozinho quando atinge as posições finais do seu curso (totalmente aberta ou totalmente fechada). Consultando a datasheet do optotriac e por tentativa e erro de cálculos sucessivos chegou-se à conclusão que a corrente no circuito isolado (5V em série com a resistência de 220Ω e LED) não iria exceder os 18mA, sendo uma corrente possível para a saída do microcontrolador, que não deve exceder 40mA.
O sinal DIR permite definir o sentido do movimento da persiana (subir ou descer), alterando a fase a que esta se encontra ligada (em relação a um neutro comum) através do relé K1. Admitindo que o MOSFET não absorve corrente praticamente nenhuma é fácil verificar que a máxima corrente que circula no ramo à esquerda da sua *gate* é de

50µA (valor muito maior que os 10nA máximos da *gate*). Esta corrente é, por uma grande margem, suportada pelo microcontrolador.

2.2 Botões

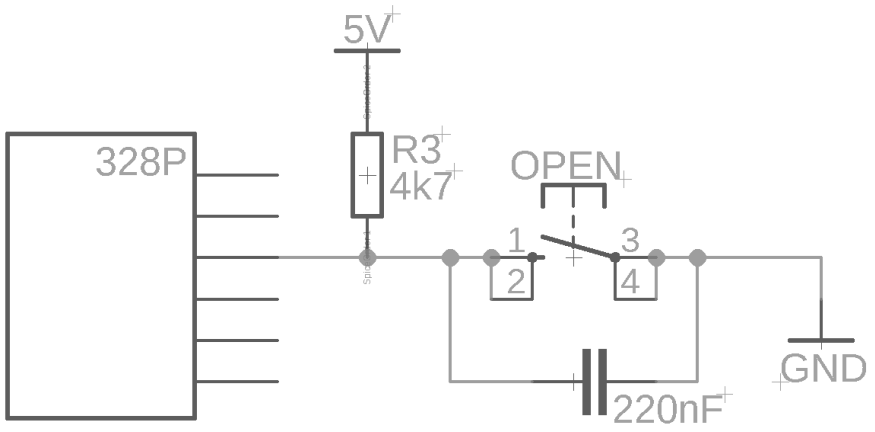


Figura 2 - Circuito dos botões. O ATMega328p econtra-se representado numa forma simplista e meramente representativa

Na figura está representado apenas o botão de abertura, mas o botão de fecho segue a mesma lógica. Os botões foram dimensionados de forma a que circule uma corrente praticamente nula quando não se carrega no botão, e uma corrente reduzida no tempo em que o botão está pressionado. Não estando o botão carregado pode-se considerar que não há corrente em qualquer sentido já que o microcontrolador apresenta no seu circuito interno os mesmos 5V que estão ligados à resistência de *pull-up*. Quando o botão é carregado a corrente que circula por efeito dos 5V "externos" (provenientes do microcontrolador, do pino Vcc) com a resistência *pull-up* é de $\frac{5V}{R3}$. Arbitrando esta corrente como sendo de 1mA obtém-se um R3 de 5kΩ que será substituído por uma resistência de 4k7Ω compatível com a série E12. O condensador de 220nF vai continuar a permitir que o sinal visto na entrada do microcontrolador desça rapidamente ao carregar no botão, mas suba "lentamente" (cerca de 4ms considerando 4τ) ao largar o botão, evitando *bouncing* do sinal.

Capítulo 3
Software

3.1 - Máquina de Estados

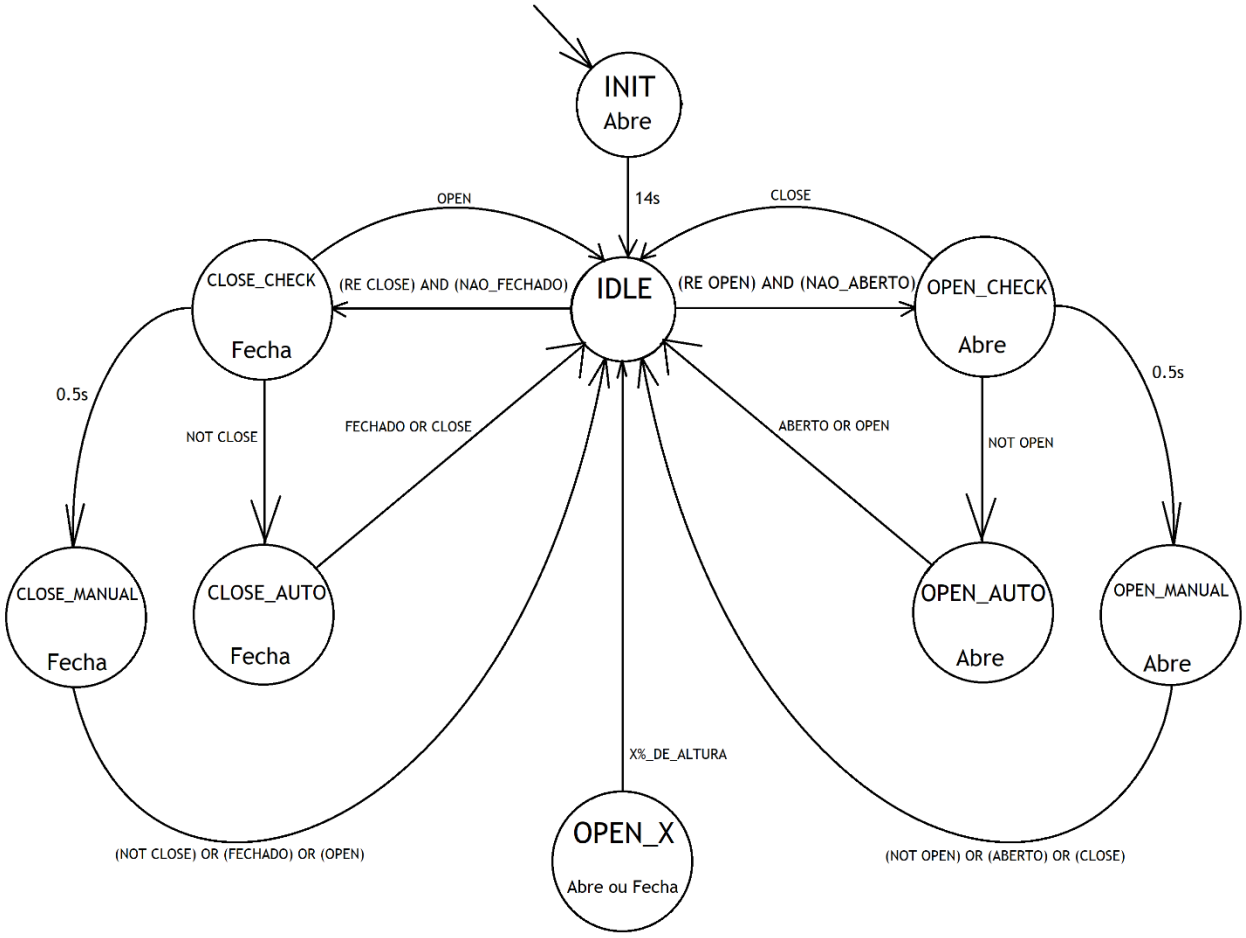


Figura 3 – Versão simplificada da máquina de estados implementada.

Esta foi a máquina de estados que implementámos por código. OPEN e CLOSE referem-se aos respetivos botões e FECHADO, NÃO_FECHADO, ABERTO e NÃO_ABERTO representam as condições respetivas, sendo meramente representativas (em código serão substituídas por condições mais explícitas). Não está representado nesta máquina de estados o comando por porta série, nem as transições respetivas à troca de direção aquando do modo automático para não tornar a figura demasiado confusa. Da troca de direção no modo automático de abertura, surgiria uma seta de OPEN_AUTO até CLOSE_CHECK com a condição CLOSE. Analogamente, para trocar a direção no modo automático de fecho, surge uma seta de CLOSE_AUTO para OPEN_CHECK com a condição OPEN. Para a comunicação por porta série avalia-se constantemente se foi

recebido algum caractere, e, se se recebeu, decide-se qual o próximo estado. Isto podia ser realizado desenhando para todos os estados três setas que iriam desde esse estado até OPEN_AUTO, CLOSE_AUTO ou OPEN_X dependendo do caractere o que seria pouco prático. Para OPEN_X a saída não é uma saída predeterminada, mas sim uma saída dependente da altura atual em relação à altura pretendida (dependendo do comando recebido), esta dependência não está explícita na máquina de estados, mas é abordada mais à frente.

3.2 – Código

O código completo está nos Anexos, neste capítulo iremos explicar cada parte do código individualmente. Este código, tal como a grande maioria de códigos com o intuido de automatizar, é constituído de uma secção de inicialização que estabelece uma base para que o programa possa funcionar (estabelecer entradas/saídas, configurar timers e comunicação por porta série) seguido de um ciclo que corre infinitamente que contém a aplicação propriamente dita. Este ciclo segue uma lógica de máquina de estados, usando uma variável que contém um número simbólico indicativo do estado atual, que por sua vez representa em que situação se encontra a persiana. Qualquer comando por porta série irá forçar o estado a "obedecer" ao comando, independentemente do estado atual (com exceção do estado de inicialização), sendo assim que se estabelece a prioridade a estes comandos. As saídas (motor e sua direção) são controladas no início de cada estado com o fim de evitar uma estrutura "*switch*" separada para o efeito.

3.2.1 – Variáveis

Segue uma tabela com todas as variáveis utilizadas e uma breve descrição da sua função:

Nome	Tipo	Descrição
OpenBtn	1 byte (uint8_t)	Guarda o valor do estado do botão de abertura. Se o botão for carregado esta variável irá ter o valor 1 durante esse ciclo, e 0 no caso contrário.
CloseBtn	1 byte (uint8_t)	Igual a OpenBtn, exceto que se aplica para o botão de fecho
RE_OpenBtn	1 byte (uint8_t)	Rising edge da variável OpenBtn (só permanece a 1 durante um ciclo de main, após o qual retorna a 0 obrigatoriamente, sendo necessário uma nova transição 0->1 da variável OpenBtn para esta variável voltar a 1)
RE_CloseBtn	1 byte (uint8_t)	Igual a RE_OpenBtn, exceto que se aplica à variável OpenBtn
USB_input	1 byte (uint8_t) volatile	Guarda o caractere recebido por porta série. Esta variável é processada no início de cada ciclo retornando a 0 (NULL) assim que for processada.
state	1 byte (uint8_t)	Variável de estado. Contém um número inteiro representativo do estado em que se encontra (os números são substituídos por nomes simbólicos no código, com o fim de facilitar a compreensão da evolução da máquina de estados)
height	2 bytes (uint16_t) volatile	Guarda a "altura" atual. Aqui, o termo "altura" não deve ser interpretado literalmente, já que a altura da persiana não é proporcional ao tempo em que o motor está ligado. Na verdade, a variável guarda o tempo que resta no caso de se pretender fechar completamente a persiana.
height_reference	2 bytes (uint16_t)	Guarda a "altura" de referência que deverá ser necessária atingir no caso de se enviar um comando por porta série que pretenda abrir a persiana entre 10% e 90%
check_delay	2 bytes (uint16_t) volatile	Temporizador com dois propósitos. Inicialmente contém um valor alto o suficiente para a persiana fechar completamente. Após isso este temporizador será sempre usado para cronometrar o tempo que se está a premir qualquer um dos botões a fim de distinguir entre um clique rápido e um clique lento

Tabela 1 – Variáveis utilizadas no código e uma breve descrição da sua função. Variáveis cujo tipo contenha *volatile* são variáveis alteradas fora do main() que poderão ser alteradas a qualquer momento, sendo necessário passar essa informação ao compilador.

3.2.2 – Inicializações

config_io() :
Através da função config_io(void) configurámos os pinos de entrada e de saída do arduíno. Como saídas definimos os pinos que controlavam o Motor e a Direção da persiana, como entradas definimos os pinos dos botões de abertura e de fecho da persiana. Ainda nesta função, coloca-se o pino do Motor a 1 de modo a garantirmos que, inicialmente, ele se encontra desligado.

config_timer2():
Esta função irá configurar o modo de operação do temporizador, a sua contagem inicial, o *prescaler* a utilizar e limpar qualquer pedido de interrupção pendente.
Para se manter, constantemente, noção do tempo decorrido, torna-se necessário usufruir dos temporizadores inerentes ao ATMega328p. Para tal tem de se definir dois parâmetros: a frequência de oscilação do cristal oscilador, e o tempo que se deseja (para haver interrupção). O cristal oscilador utilizado no Arduino é de 16MHz, e pretendemos uma base de tempo de 1ms (desde que os tempos pretendidos sejam múltiplos da base de tempo, em teoria, qualquer valor de base de tempo serve). O número de oscilações é constante e inalterável para uma dada frequência e tempo de interrupção, no entanto, podemos alterar a maneira como as contamos através de *prescalers* que contam de k em k oscilações com base num valor por nós definido. Existe um *prescaler* geral (CP), que afeta o sinal gerado pelo cristal para uma grande parte do ATPMega328p, um *prescaler* específico (TP) para o temporizador 0 e 1 (conjuntamente), e um *prescaler* específico (TP) para o temporizador 2. Ainda mais, cada um destes temporizadores têm uma capacidade de dados limitada de quantas oscilações conseguem contar, sendo essa capacidade de 1 byte para os temporizadores 0 e 2 (contam até 255), e de 2 bytes para o temporizador 1 (conta até 65535). Assim o número de oscilações a contar do ponto de vista de cada temporizador (CNT) irá depender do CP e TP escolhidos, bem como da base de tempo. Havia alguma liberdade com a escolha da base de tempo para o timer, no entanto, e prevendo que controlar a persiana através do seu tempo de subida iria gerar um erro significativo, escolheu-se uma base de tempo de 1ms para atenuar este erro. Construindo uma tabela de cálculo que nos indica o CNT a utilizar com base em diversas combinações de CP e TP disponíveis, podemos analisar qual o melhor CNT a utilizar e qual o melhor temporizador para esse mesmo CNT:

CP\TP	1	8	32	64	128	256	1024
1	**16000**	**2000**	impossível	250	125	62.5	15.625
2	**8000**	**1000**	250	125	62.5	31.25	7.8125

Tabela 2 – Combinações possíveis de contagens dependendo do prescaler geral (CP) e do prescaler do *timer* (TP). Números a negrito só são possíveis no timer 1, números vermelhos são contagens fracionárias e portanto geram um erro.

CP\TP	1	8	32	64	128	256	1024
1	0.00	0.00	*N/A*	0.00	0.00	0.80	4.00
2	0.00	0.00	0.00	0.00	0.80	0.80	10.40

Tabela 3 - Erros associados aos respetivos valores da Tabela 1, em percentagem.

Na tabela de cima é apresentado o valor de CNT a utilizar para a combinação de TP e CP respetiva, e na tabela de baixo o respetivo erro para essas combinações. Este erro advém do facto que é impossível contar frações de impulsos. Forçando CP=1, temos 4 possibilidades utilizando o temporizador 2 de 8 bits. Escolhendo o menor CNT (já que menos oscilações implica menos transições e, consequentemente, menos gasto de energia) sem que haja qualquer erro, optamos pela combinação: TP = 128; CP = 1; CNT = 125 impulsos. O temporizador irá contar até ocorrer *overflow* pelo que podemos impor a condição da contagem de 125 impulsos forçando o seu contador a iniciar em 130 (255-125 = 130) em vez de iniciar em 0.

init_usart ():
Para utilizarmos comunicação série utilizámos um método de comunicação assíncrona, e assim, tornou-se necessário definir uma *Baudrate* que foi arbitrada como sendo de 57600. Nesta função, configurámos de modo a trabalharmos no modo normal de amostragem estabelecendo 16 amostras por cada bit recebido. Sendo a frequência do cristal do ATMega328p de 16MHz, isto traduz-se num registo de:

$$UBBR = \frac{16M}{16 * 57600} - 1 \cong 16$$

$$BAUDRATE = \frac{16M}{16 * (16 - 1)} = 58823{,}52941\ bps$$

Ambos calculados através de fórmulas da datasheet do ATMega328p, sendo esta última *baudrate* a que deveras se verifica e terá um erro de 2,124% que não será muito significativo, tendo em conta que não serão usados bits de paridade e apenas 1 stop bit com 8 bits de dados (9 bits no total) que corresponde a um erro máximo admissível de cerca de 5,56%.
Também nesta função de configuração são ativos os registos de leitura e escrita bem como o de interrupção por leitura, ou seja, cada vez que é lido um valor é gerado o respetivo pedido de interrupção.

3.2.3- Interrupções

Este código gera dois tipos de interrupções, ISR (TIMER2_OVF_vect) é gerada a cada 1ms devido ao overflow do temporizador 2 e a ISR (USART_RX_vect) é gerada sempre que recebe dados pela porta série.
A rotina de interrupção do temporizador 2 irá decrementar a variável "check_delay" até esta atingir 0 (age como temporizador) e irá aumentar ou reduzir a variável "height" conforme o motor esteja a subir ou a descer (se o motor estiver desligado a variável permanece inalterada).
A interrupção ISR (USART_RX_vect) guarda o valor recebido, que gerou a própria interrupção, numa variável e devolve-o, notando que a devolução é meramente por questões informativas ao utilizador). Esta variável será avaliada no main() de forma a averiguar se é um input válido e qual o comando a que lhe corresponde, de entre os seguintes: ‘u’ que deve abrir completamente a persiana; ‘g’ que deve colocar a persiana numa posição que a parte inferior da persiana não toca no chão mas a persiana não está aberta, ou seja, apenas com as tábuas separadas; e ‘0’ ~ ‘9’ que deve abrir x% (em que 0% corresponde a fechar a persiana na totalidade).Na interrupção ISR (TIMER2_OVF_vect), o contador será colocado a 125 contagens do seu valor de *overflow*, de forma a preservar as 125 contagens necessárias a interrupções periódicas de 1 ms. Isto é feito atribuindo diretamente à variável do contador o valor referido. Ainda nesta rotina de interrupção, decrementamos a variável “check_delay” até esta atingir 0, ou seja, age como temporizador e irá aumentar ou reduzir a variável “height”, correspondente á posição da persiana, que irá aumentar ou reduzir conforme o motor esteja a subir ou a descer, se o motor estiver desligado a variável permanece inalterada).

3.2.4 – Switch – máquina de estados

A estrutura geral de código associada à máquina de estados implementada é como a que se segue:

```
> switch (estado) {
>   ações_estado1();
>
>   case 0:
>     if (condições_transição1) {
>       ações_transição1();
>       estado = estado_transição1;
>     }break;
>     else if (condições_transição2) {
>       ...
>   case 1:
>       ...
>   ...
> }
>
```

3.2.5 – Processamento de comandos recebidos (porta série)

Os comandos por porta série têm um efeito direto sob a máquina de estados, mas o seu processamento não pertence à máquina de estados propriamente dita. No início de cada ciclo verifica-se se a variável *USB_input* tem um valor diferente de 0 (NULL), se tiver, verifica-se qual o comando recebido, decidindo e forçando o estado atual a corresponder ao comando recebido. A variável é posteriormente forçada a NULL para evitar leituras repetidas do mesmo comando. Os comandos possíveis e os respetivos estados forçados são:

Comando (ASCII)	Estado forçado
‘u’	OPEN_AUTO
‘0’	CLOSE_AUTO
‘1’,’2’,’3’,…,’9’ e ‘g’	OPEN_X

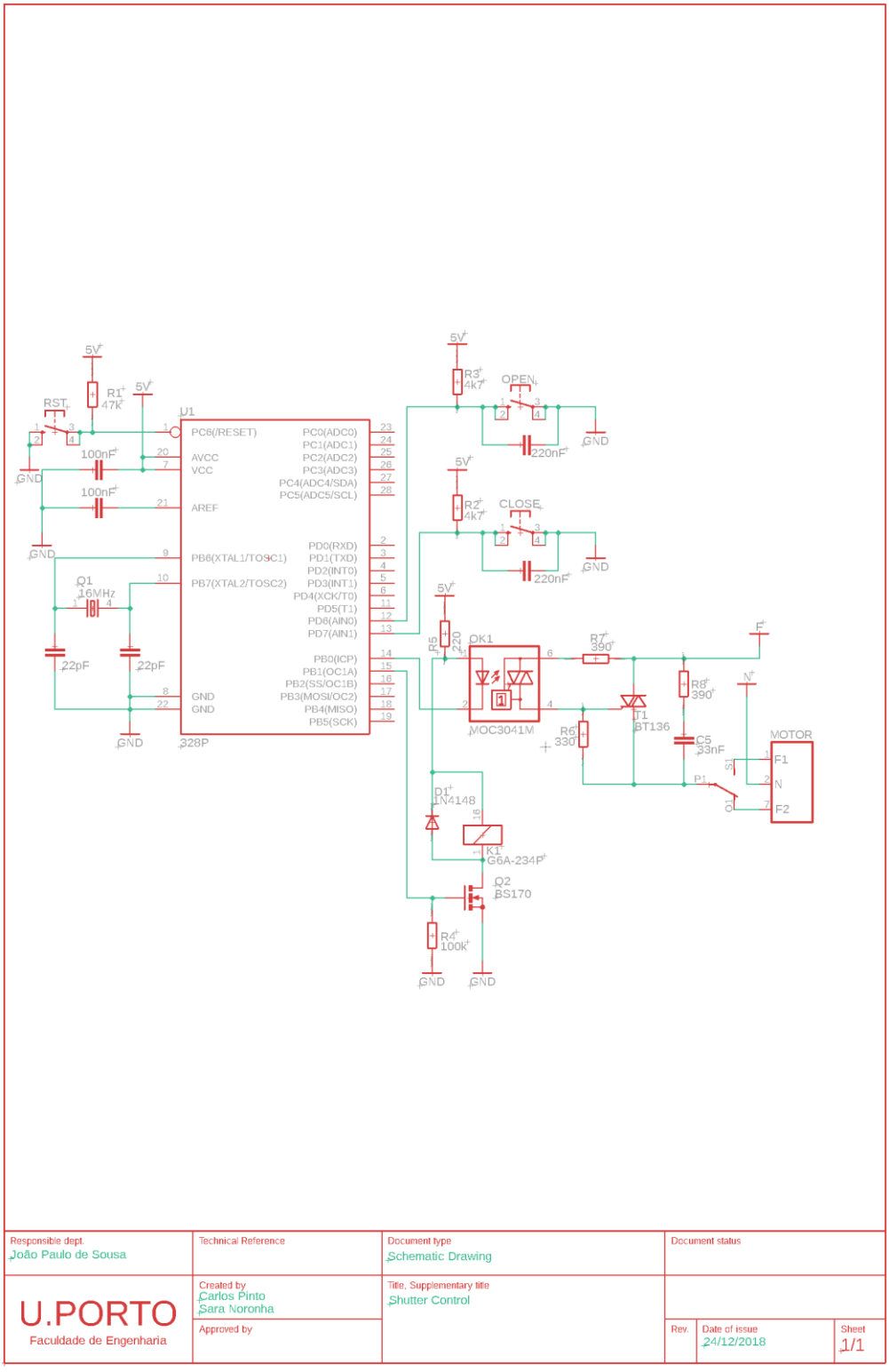
Tabela 4 - Estados forçados dependendo do comando recebido. Ao forçar o estado OPEN_X também é estabelecida uma altura de referência na variável *height_reference* que irá corresponder ao comando recebido. O comando ‘g’ corresponde a separar apenas as tábuas da persiana.

Conclusão

Com este trabalho consolidámos alguma da matéria lecionada na unidade de curricular Sistemas Baseados em Microprocessadores como Comunicação em Série – USART e Timers.
Na avaliação deste trabalho, não estava concebido o cabeçalho na sua totalidade, que foi posteriormente completado.
Também na avaliação discutiram-se ideias que podiam vir a melhorar este projeto, como por exemplo, a alteração da maneira como foi calculada o quanto devia abrir a persiana, quando se usa na comunicação porta série percentagens. Usaram-se frações do tempo inicialmente cronometrado da abertura na sua totalidade e foram tidas em conta as diferentes velocidades de subida da persiana por efeito do seu enrolamento no cilindro do motor. Para corrigir isto, podia-se cronometrar tempos independentes para cada percentagem, ou calcular, teoricamente, qual seria a velocidade de subida da persiana dependendo do raio formado em torno do cilindro, assumindo uma rotação constante.
Pretendia-se acrescentar uma hora fixa em memória não volátil em que a persiana iria abrir/fechar automaticamente a essa hora, mas, por falta de tempo, esta funcionalidade não ficou incluída no trabalho final.

Anexos

Esquema do circuito completo



Código final (main.c)

```
/*
 * main.c
 *  Um programa de controlo de persiana com comunicação por porta
 *  série.
 *
 *  Objetivo:
 *  Controlar uma persiana usando dois botões (cima e baixo), se
 *  se carregar rapidamente num dos botões a persiana abre/fecha
 *  completamente (modo automático). Se se carregar lentamente num
 *  dos botões a persiana abre/fecha até se deixar de carregar no
 *  respetivo botão (modo manual). Alternativamente, por
 *  computador (ou qualquer dispositivo capaz de enviar tramas por
 *  porta série para o Arduino com código ASCII) poder controlar o
 *  nível de abertura da persiana (fechada, aberta, apenas separar
 *  tábuas ou abrir até uma determinada percentagem).
 *
 *  Pormenores contextuais:
 *   -No tempo em que ainda não se decidiu entre um clique
 *  "rápido" e um clique "lento" a persiana já se encontra em
 *  movimento.
 *   -Estando a persiana a abrir num sentido, em modo automático,
 *  se se carregar no botão de sentido oposto, a persiana obedece
 *  mudando de direção.
 *   -Estando a persiana a abrir num sentido, em modo automático,
 *  se se carregar no botão do mesmo sentido, a persiana para.
 *   -Estando a persiana a abrir num sentido, em modo manual, se
 *  se carregar no botão de sentido oposto, a persiana para e
 *  aguarda até que um novo botão seja novamente carregado.
 *   -Os comandos por porta série têm prioridade absoluta em
 *  relação aos botões (comandos nunca serão ignorados nem
 *  adiados). E mesmo estando numa operação causada por comando
 *  de porta série, um novo comando será sempre imediatamente
 *  respondido.
 *   -Considerou-se que o tempo que a persiana demora a abrir
 *  é de 13,2s (cronometrado) e que a sua velocidade de subida
 *  permanece constante, o que não é verdade e gera um erro
 *  cumulativo.
 *   -Considerou-se que a persiana consegue mudar de direção
 *  enquanto o motor está ligado, sem qualquer problema e sem
 *  atrasos. Infelizmente, como a mudança de direção implica uma
 *  mudança brusca da fase a que a persiana está ligada, esta
 *  troca pode, eventualmente, estragar a persiana. Deve-se,
 *  assim, evitar estas transições perigosas.
 *
 *  Estrutura do código:
 *   Este código, tal como a grande maioria de códigos
 *   com o intuído de automatizar, é constituído de uma secção de
 *   inicialização que estabelece uma base para que o programa
 *   possa funcionar (estabelecer entradas/saídas, configurar
 *   timers e comunicação por porta série) seguido de um ciclo que
 *   corre infinitamente que contém a aplicação propriamente dita.
 *   Este ciclo segue uma lógica de máquina de estados, usando uma
```

```
*   variável que contém um número simbólico indicativo do estado
*   atual, que por sua vez representa em que situação se encontra
*   a persiana. Qualquer comando por porta série irá forçar o
*   estado a "obedecer" ao comando, independentemente do estado
*   atual (com exceção do estado de inicialização), sendo assim
*   que se estabelece a prioridade a estes comandos. As saídas
*   (motor e sua direção) são controladas no início de cada estado
*   com o fim de evitar uma estrutura "switch" separada para o
*   efeito.
*
*  Timer:
*   Havia alguma liberdade com a escolha da base de tempo para o
*   timer, no entanto, e prevendo  que controlar a persiana
*   através do seu tempo de subida iria gerar um erro
*   significativo, escolheu-se uma base de tempo de 1ms para
*   atenuar este erro.
*   Com uma base de 1ms a solução de maior prescaler, TP, e
*   prescaler de "Clock" a 1 (para evitar afetar outros timers e
*   processos), é com uma contagem de 125 (obtido por Excel):
*    CP * TP * CNT = Fcpu * Tint
*    1  * 128* 125 = 16M  * 1m
*   Não havendo outras exigências provenientes do timer, será
*   poupado o timer 1, e implementado o timer 2.
*   Será implementado o modo normal cujo contador aumenta até
*   ocorrer overflow, gerando um pedido de interrupção e
*   reinicializando a 0. Em cada interrupção o contador será
*   colocado a 125 contagens do seu valor de overflow, de forma
*   a perservar as 125 contagens necessárias a interrupções
*   periódicas de 1 ms. Isto é feito atribuindo diretamente à
*   variàvel do contador o valor referido.
*   A rotina de interrupção do timer irá decrementar a variável
*   "check_delay" até esta atingir 0 (age como temporizador) e
*   irá aumentar ou reduzir a variável "height" conforme o motor
*   esteja a subir ou a descer (se o motor estiver desligado a
*   variável permanece inalterada).
*
*  Comunicação série:
*   Utilizando um método de comunicação assíncrona, torna-se
*   necessário definir uma Baudrate, que foi arbitrada como
*   sendo de 57600. Vai-se utilizar o modo normal de amostragem,
*   estabelecendo 16 amostras por cada bit recebido. Sendo a
*   frequência do cristal associado ao ATMega328p de 16MHz, isto
*   traduz-se num registo de UBRRO (seguindo fórmula da
*   datasheet):
*    UBBRO = 16M/(16*57600)-1 = 16.361 (arredondando) = 16
*    BAUDRATE = 16M/(16*(16-1)) = 58823,52941 bps
*   Ou seja, um erro de 2,124% que não será muito significativo
*   tendo em conta que não serão usados bits de paridade, e
*   apenas 1 stop bit com 8 bits de dados (9bits no total) que
*   corresponde a um erro máximo admissível de cerca de 5,56%.
*   São ativos os registos de leitura e escrita bem como de
*   interrupção por leitura. Sempre que é lido um valor é gerado
*   o respetivo pedido de interrupção que simplesmente guarda o
```

```
 *   valor recebido numa variável e devolve-o (a devolução do valor
 *   apenas serve para questões informativas ao utilizador). Esta
 *   variável será avaliada no main() de forma a averiguar se é um
 *   input válido e qual o comando a que lhe corresponde, de entre
 *   os seguintes:
 *    'u'-abrir completamente
 *    'g'-colocar as tábuas separadas sem abrir a persiana
 *    '0'~'9'- abrir até x% (0% corresponde a fechar a persiana)
 *
 *  Debug:
 *   O debug não é da nossa autoria tendo sido utilizado para o
 *   efeito a biblioteca "serial.c" da autoria do docente João
 *   Paulo Sousa. Esta biblioteca redireciona a stream stdout
 *   colocando a componente ".put" de FDEV_SETUP_STREAM para a nova
 *   função "usart_putchar()" que escreve, sempre que possível, um
 *   caractere da string indicada por "printf()". A flag de
 *   FDEV_SETUP_STREAM é colocada em modo de escrita
 *   ("_FDEV_SETUP_WRITE"). (".get" não terá nenhum valor pois
 *   apenas se pretende escrever para o PC)
 *   A função "printf_init()" limita-se a atualizar o valor da
 *   variável stdout para a nova configuração descrita.
 *   Para ativar o modo Debug basta definir a constante DEBUG na
 *   linha de código 143.
 *   O código debug leva muito tempo a correr em relação ao resto
 *   do programa, e é apenas usado aquando do teste da persiana
 *   na fase de desenvolvimento.
 *
 *  Cabeçalho creado em: 26/11/2018 (pós avaliação presencial)
 *  Código creado em: 15/11/2018
 *      Autores: Carlos Manuel Santos Pinto
 *               Maria Sara Delgadinho Noronha
 */

#include <avr/interrupt.h>
#include "serial.h"

// DEBUG mode
//#define DEBUG
// Nomes simbolicos para os estados
#define INIT 0 // Inicialização
#define IDLE 1 // Aguarda Comandos
#define CLOSE_CHECK 2 // Fecha e decide entre manual e automático
#define OPEN_CHECK 3 // Abre e decide entre manual e automático
#define OPEN_AUTO 4 // Abre automaticamente
#define CLOSE_AUTO 5 // Fecha automaticamente
#define CLOSE_MANUAL 6 // Fecha manualmente
#define OPEN_MANUAL 7 // Abre manualmente
#define OPEN_X 8 // Abre/fecha até X% da altura máxima (altura de referência)
#define ILLEGAL 255 // Para estados imprevistos
```

```
#define T2BOTTOM 255-125 // Valor inicial de contagem de timer 2 (ver função
"config_timer2()")

#define MOTOR PB0 // Posição respetiva ao pino do motor (ativo a 0)
#define DIR PB1 // Posição respetiva ao pino da direção do motor (0 vai para
cima, 1 vai para baixo)
#define CLOSE PD6 // Posição respetiva ao pino do botão de fecho (ativo a 0)
#define OPEN PD7 // Posição respetiva ao pino do botão de abertura (ativo a 0)

#define CHECK_TIME 500 // 0.5s para distinguir entre clique rápido e lento
#define INIT_TIME 14000 // 14s para fechar totalmente (garantidamente)

#define MAX_HEIGHT 13200 //13.2s para chegar à máxima altura (cronometrado -
sujeito a erro)
#define OPEN_TIME 2500 // Corresponde ao tempo que a persiana demora a começar
a abrir (tábuas deixam de tocar na base, também foi cronometrado)
#define OPEN_10 (MAX_HEIGHT-OPEN_TIME)/10 // Valor relativo (10%) de abertura
descontando o tempo de abertura definido na linha anterior

#ifndef F_CPU
#define F_CPU 16000000ul // 16MHz de frequência de relógio do processador
(para estabelecer a Baudrate)
#endif

#define BAUD 57600ul // Baudrate de 57600 simbolos/s
#define UBBR_VAL ((F_CPU/(BAUD*16))-1) // 16 amostras por símbolo (modo
normal)

uint8_t OpenBtn = 0; // Variável auxiliar de verificação (Botão de abertura
pressionado -> 1)
uint8_t CloseBtn = 0; // Variável auxiliar de verificação (Botão de fecho
pressionado -> 1)
uint8_t RE_OpenBtn = 0; // Rising Edge de OpenBtn
uint8_t RE_CloseBtn = 0; // Rising Edge de CloseBtn
volatile uint8_t USB_input = 0; // Variável auxiliar que vai guardar o
caracter recebido por porta série
uint8_t state = INIT; // Estado atual
unsigned int height_reference = 0; // Altura de referência (apenas pertinente
no estado OPEN_X)
volatile unsigned int height = MAX_HEIGHT; // Altura atual, inicia no topo
volatile unsigned int check_delay = INIT_TIME; // Tempo de espera na decisão
entre clique rápido e lento. Também é
                                                // usado como timer na
inicialização para garantir que a persiana fecha
#ifdef DEBUG
uint8_t printfstate = 254; // Último estado impresso por printf (apenas
pertinente no caso de debug)
#endif
```

```
/* Configura pinos de entrada/saída */
void config_io (void){
  DDRB |= ((1<<MOTOR) | (1<<DIR)); // configura os pinos respetivos ao motor e
sua direção como saídas
  DDRD &= (~(1<<CLOSE) & ~(1<<OPEN)); // configura os pinos respetivos aos
botões de abertura/fecho como entradas

  PORTB |= (1<<MOTOR); // Garante que o motor está, inicialmente, desligado
}


/* Configura timer 2 para contar em ciclos de 1ms:
 * Frequência do CPU = 16MHz;
 * TP = 128; CP = 1; CNT = 125 impulsos (255-130);
 * onde 130 é o valor inicial de contagem para que
 * sejam contados 125 impulsos até overflow.
 * Tempo entre interrupções é dado por:
 *   128*125/16M = 1ms */
void config_timer2 (void){
  TCCR2B = 0; // Para o timer
  TIFR2 |= (7<<TOV2); // Desliga quaisquer flags que estejam ativas
  TCCR2A = 0; // Modo de contagem normal
  TCNT2 = T2BOTTOM; // Posiciona o inicio da contagem no valor estabelecido em
T2BOTTOM
  TIMSK2 = (1<<TOIE1); // Permite interrupção por overflow
  TCCR2B = 5; // Inicia o timer com um prescaler : TP=128
}

void init_usart(){
      // Configuração do conjunto de bits para determinar frequência de
transição entre bits
      UBRR0 = UBBR_VAL;
      /* Significa que cada bit será amostrado 16 vezes
       * Que por sua vez significa que teremos uma
       * Baudrate efetiva de 58823.52941 Bps */

      UCSR0C = (3<<UCSZ00) // Cada segmento terá 8 bits de informação útil
(data bits),
               | (0<<UPM00)  // sem paridade,
               | (0<<USBS0); // e 1 bit de paragem

      UCSR0B = (1<<TXEN0) | (1<<RXEN0) | (1<<RXCIE0); // Permite leitura,
escrita e interrupção após leitura
}

ISR (USART_RX_vect) { // Sempre que recebe dados por porta série
 USB_input = UDR0; // ATMega recebe os dados do PC e guarda-os
 UDR0 = USB_input; // Envia os dados que recebeu, de volta para o PC
}
```

```
ISR (TIMER2_OVF_vect){ // Interrupção gerada a cada 1ms
  TCNT2 = T2BOTTOM; // atualiza valor inicial de contagem do timer 2

  if ( !(PINB & (1<<MOTOR)) && !(PINB & (1<<DIR)) && (height)){ // se o motor
estiver a fechar
    height--; // decrementa altura
  }
  else if ( !(PINB & (1<<MOTOR)) && (PINB & (1<<DIR)) && (height <
MAX_HEIGHT)){ // se o motor estiver a abrir
    height++; // incrementa altura
  }

  if (check_delay){ // se check_delay ainda não atingiu 0
    check_delay--; // decrementa
  }
}



int main(){

  init_usart(); // Configura a comunicação por porta série
  config_io(); // Configura pinos de entrada e saída
  config_timer2(); // Configura timer 2
  sei(); // Ativar bit geral de interrupções, permitindo interrupções em geral

  #ifdef DEBUG
    printf_init();
    printf("\n_____________________|DEBUG ON|_____________________\n");
  #endif


  while(1){ // Ciclo infinito (Loop)

    // Leitura de entradas no mesmo instante
    RE_CloseBtn = (!(PIND & (1<<CLOSE))) && (!CloseBtn); // Ativo no flanco
ascendente do botao de abertura
    CloseBtn = !(PIND & (1<<CLOSE)); // Botao de fecho (ativo a 1)
    RE_OpenBtn = (!(PIND & (1<<OPEN))) && (!OpenBtn); // Ativo no flanco
ascentende do botao de abertura
    OpenBtn = !(PIND & (1<<OPEN)); // Botao de abertura (ativo a 1)
```

```
    /* A porta série tem prioridade sobre os botões, portanto assim que algo é
lido, é
     * processado o que foi recebido e a máquina de estados é forçada ao
estado adequado */
    if(USB_input!=0){ // Se algo foi lido por porta série força a máquina de
estados a responder de acordo
      if (INIT!=state){
        if ('u' == USB_input){ // Se se premiu "u", abre completamente
          state = OPEN_AUTO; // Abre (completamente) em modo automático
        }
        else if ('0' == USB_input) { // Se se premiu "0", fecha completamente
          state = CLOSE_AUTO; // Fecha (completamente) em modo automático
        }
        else if (USB_input>'0' && USB_input<='9'){ // Foi premido um número
que não zero
          height_reference = OPEN_10*(USB_input-48)+OPEN_TIME; // Toma valores
desde 10% a 90% de abertura, dependendo da tecla premida
          state = OPEN_X; // Abre/fecha até height_reference
        }
        else if (USB_input == 'g'){ // Se se premiu "g", separa as tábuas sem
abrir a persiana
          height_reference = OPEN_TIME; // Altura de abertura efetiva da
persiana
          state = OPEN_X; // Abre/fecha até ficar com as tábuas separadas
        }
      }

      USB_input = 0; // Reset da variável de leitura
    }

    switch (state){
      case INIT: // 0 - Inicialização (fecha totalmente a persiana e ignora
comandos do utilizador)
        PORTB &= ~(1<<MOTOR); // Liga motor
        PORTB |= (1<<DIR); // com direção para cima (abre)

        if (!check_delay){ // Se já passou tempo de inicialização (check_delay
é usado como timer de inicialização apenas nesta linha)
          state = IDLE; // para de fechar e aguarda comandos do utilizador
        }break;
```

```
      case IDLE: // 1 - Espera por qualquer ação (Inativo)
        PORTB |= (1<<MOTOR); // desliga motor

        if (RE_CloseBtn && height ){ // Quer-se fechar e ainda não está
fechado nem se está a carregar no botão de abrir
          state = CLOSE_CHECK; // Fecha persiana e...
          check_delay = CHECK_TIME; //...inicializa contagem do tempo que se
mantém o botão carregado
        }
        else if (RE_OpenBtn && (MAX_HEIGHT != height) ){ // Quer-se abrir e
ainda não está aberto
          state = OPEN_CHECK; // Abre persiana e...
          check_delay = CHECK_TIME; // ...inicializa contagem do tempo que se
mantém o botão carregado
        }break;


      case CLOSE_CHECK: // 2 - Fecha e verifica quanto tempo se prime o botão
        PORTB &= ~(1<<MOTOR); // Liga motor
        PORTB &= ~(1<<DIR); // com direção para baixo (fecha)

      if (!height){ // Se já fechou completamente
        state = IDLE; // para de fechar
      }
        else if (!CloseBtn){ // Se já não se está a carrergar no botão
(implica que foi um clique rápido)
          state = CLOSE_AUTO; // fecha em modo automático
        }
        else if (!check_delay){ // Se já passou o tempo (implica que foi um
clique lento)
          state = CLOSE_MANUAL; // fecha em modo manual
        }
        else if (OpenBtn) { // Se se carregar no botão de abertura
          state = IDLE; // Para de fechar
        }break;
```

```
      case OPEN_CHECK: // 3 - Abre e verifica quanto tempo se prime o botão
        PORTB &= (~(1<<MOTOR)); // Liga motor
        PORTB |= (1<<DIR); // com direção para cima (abre)

        if (MAX_HEIGHT == height){ // Se já abriu completamente
          state = IDLE; // para de abrir
        }
        else if (!OpenBtn){ // Se já não se está a carrergar no botão (implica
que foi um clique rápido)
          state = OPEN_AUTO; // abre em modo automático
        }
        else if (!check_delay){ // Se já passou o tempo (implica que foi um
clique lento)
          state = OPEN_MANUAL; // abre em modo manual
        }
        else if (CloseBtn){ // Se se carregar no botão de fecho
          state = IDLE; // Para de abrir
        }break;


      case OPEN_AUTO: // 4 - Abre até a persiana ficar completamente aberta
        PORTB &= (~(1<<MOTOR)); // Liga motor
        PORTB |= (1<<DIR); // com direção para cima (abre)

        if (MAX_HEIGHT == height || OpenBtn){ // Se já abriu completamente ou
voltou-se a carregar no botão de abrir
          state = IDLE; // para de abrir
        }
        else if(CloseBtn){ // Se se carregou no botão de fecho
          state = CLOSE_CHECK; // Fecha persiana e...
          check_delay = CHECK_TIME; //...inicializa contagem do tempo que se
mantém o botão carregado
        }break;



      case CLOSE_AUTO: // 5 - Fecha até a persiana ficar completamente fechada
        PORTB &= ~(1<<MOTOR); // Liga motor
        PORTB &= ~(1<<DIR); // com direção para baixo (fecha)

        if ( (!height) || CloseBtn){ // Se já fechou completamente ou voltou-
se a carregar no botão de fechar
          state = IDLE; // para de fechar
        }
        else if (OpenBtn){ // Se carregar no botão de abrir posteriormente a
persiana volta a abrir
          state = OPEN_CHECK; // Abre persiana e...
          check_delay = CHECK_TIME; // ...inicializa contagem do tempo que se
mantém o botão carregado
        }break;
```

```
      case CLOSE_MANUAL: // 6 - Fecha persiana até deixar de premir o botão
(ou fechar completamente)
        PORTB &= ~(1<<MOTOR); // Liga motor
        PORTB &= ~(1<<DIR); // com direção para baixo (fecha)

        if (!CloseBtn || !height || OpenBtn){ // Se já fechou ou se deixou de
carregar no botão ou se carregou no botão de abertura
          state = IDLE; // para de fechar
        }break;


      case OPEN_MANUAL: // 7 - Abre persiana até deixar de premir o botão (ou
abrir completamente)
        PORTB &= (~(1<<MOTOR)); // Ligar motor
        PORTB |= (1<<DIR); // com direção para cima (abre)

        if (!OpenBtn || (MAX_HEIGHT == height) || CloseBtn){ // Se já abriu ou
se deixou de carregar no botão ou se carregou no botão de fecho
          state = IDLE; // para de abrir
        }break;


      case OPEN_X: // 8 - Abre/fecha até X% da altura (height_reference)
          if (height > (height_reference)){ // Se a altura atual está acima da
altura de referência
          PORTB &= ~(1<<MOTOR); // Liga motor
          PORTB &= ~(1<<DIR); // com direção para baixo (fecha)
        }
        else if (height < (height_reference)){ // Se a altura atual está
abaixo da altura de referência
          PORTB &= (~(1<<MOTOR)); // Liga motor
          PORTB |= (1<<DIR); // com direção para cima (abre)
        }
        else if (height == (height_reference)){ // Se a altura de referência
foi atingida
          state = IDLE; // Para de abrir/fechar
        }break;


      case ILLEGAL:// em casos ilegais o motor é desligado e o sistema entra
em bloqueio permanentemente
        PORTB |= (1<<MOTOR); // desliga motor
        break;

      default:// para qualquer caso fora do previsto, transita para o estado
ilegal e desliga motor
        PORTB |= (1<<MOTOR); // desliga motor
        state = ILLEGAL; // transita para estado ilegal
    }
```

```
    #ifdef DEBUG
      if (state != printfstate){
        printf("((STATE:%d; height:%d; Input:%c; check_delay:%u; OPEN:%d
CLOSE:%d  MOTOR:%d  DIR %d))\n",state, height, USB_input, check_delay, OpenBtn
,CloseBtn ,!(PINB & (1<<MOTOR)) ,(PINB & (1<<DIR)));
        printfstate = state;
      }
    #endif

  }
}
```

Código final (biblioteca “serial” – header file)

```
/*
 * serial.h
 *  Redirection of the printf stream to the AVR serial port
 *  Header file
 *  Created on: 13/09/2016
 *      Author: jpsousa@fe.up.pt
 */

#ifndef SERIAL_H_
#define SERIAL_H_

#include <stdio.h>

void usart_init(void);
int usart_putchar(char c, FILE *stream);
void printf_init(void);

#endif /* SERIAL_H_ */
```

Código final (biblioteca “serial” – library file)

```
/*****************************************************************************
 * serial.c
 *  Redirection of the printf stream to the AVR serial port
 *    1. Initialize usart @ 57600 bps (change in BAUD)
 *    2. Define the low level put_char mechanism
 *    3. Redirect the printf io stream
 *
 *  Created on: 13/09/2016
 *      Author: jpsousa@fe.up.pt (eclipse + gcc-avr)

*****************************************************************************/

#include <stdio.h>
#include <avr/io.h>          /* Register definitions*/

#define F_CPU 16000000UL               /* 16 MHz    */
#define     BAUD 57600                     /* baud rate */
#define BAUDGEN ((F_CPU/(16*BAUD))-1)  /* divider   */

void usart_init(void) {
  UBRR0 = BAUDGEN;
  UCSR0B = (1 << RXEN0) | (1 << TXEN0);
  UCSR0C = (1 << USBS0) | (3 << UCSZ00);
}

int usart_putchar(char c, FILE *stream) {
  while (!( UCSR0A & (1 << UDRE0)))
    ;
  UDR0 = c;
  return 0;
}

static FILE mystdout = FDEV_SETUP_STREAM(usart_putchar, NULL,
_FDEV_SETUP_WRITE);

void printf_init(void) {
  stdout = &mystdout;
}
```

Referências

[1] Datasheet ATmega328P, Atmel [download de 11-2016] - http://www.atmel.com/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_datasheet.pdf