First Test - Um Estudo do PIC10F200

Por Eduardo Foltran - Novembro de 2015

Índice

1 - Introdução ... 3
2 - O PIC10F200 ... 3
3 - Multiplicando as saídas ... 4
    3.1 - O registrador 74HC595 ... 4
4 - Números binários ... 5
    4.1 - Base binária ... 5
    4.2-Decimais codificados em binário (BCD) ... 6
5 - O circuito ... 6
6 - O programa ... 9
    6.1 - Declarações ... 9
    6.2 - Função main() ... 10
    6.3 - A função SendData() ... 11
    6.4 - A função Increment() ... 12
7 - O protótipo ... 13
8 - Conclusões ... 15
9 - Listagem do programa ... 15
10 - Código hexadecimal ... 19
11 - Lista de material ... 20
12 - Montagem ... 20
    12.1 - Placa ... 20
    12.2 - Posicionamento dos jumpers ... 21
    12.3 - Aspecto final da montagem ... 21
    12.4 - Imagem ampliada da montagem ... 22
    12.5 - Placas prontas para transferência térmica ... 23

1 - Introdução

Desde os doze anos de idade divido meu interesse em duas áreas próximas: eletrônica e informática. Já desenvolvi vários circuitos analógios e digitais, tanto em estado sólido quanto valvulado, programei em várias linguagens, para várias finalidades e, mais recentemente, encerrei um projeto de três anos com a nova loja virtual da Altana Tubes.

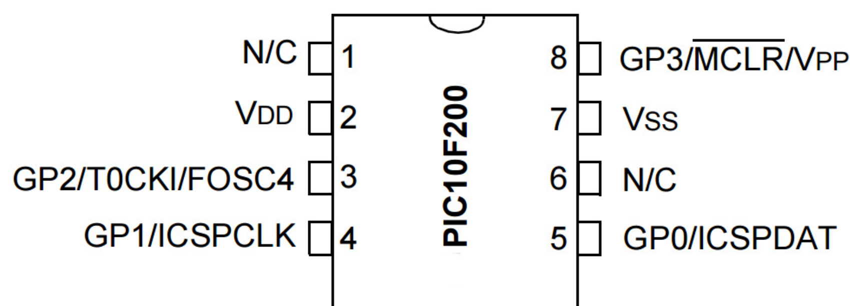
Faz quase um ano que me deparei com os microcontroladores. Saltou a minha frente a chance de juntar as duas áreas e desenvolver tanto o hardware quanto o software das coisas. Mas como nunca havia feito nada assim, vi que teria que enfrentar a tediosa e dolorosa curva de aprendizado novamente.

Resolvi iniciar meu estudo com um microcontrolador simples e um projeto com algum desafio. Escolhi desenvolver um contador de oito dígitos com display de sete segmentos multiplexado usando o PIC10F200. Com oito dígitos conta-se até 99.999.999. Isso é metade da população do Brasil. Se contado um evento por segundo, levaria mais de três anos para chegar neste valor. É óbvio que oito dígitos são exagero para um contador, mas este é um projeto didático. A intenção é demonstrar conceitos como multiplexação e números decimais codificados em binário, ou BCD (*binary coded decimals*).

Ao procurar informações sobre o microcontrolador, percebi que existe pouco material de estudo para o PIC10F200. Temos um caso sério de “matar mosca com canhão” quando se trata de microcontroladores. Como são baratos o *HandMaker* comum prefere ir direto para um controlador de 40 pinos, como o PIC16F877, ao invés de dedicar algum tempo de planejamento para usar algo menor e mais barato. Por isso decidi escrever este artigo, relatando o que aprendi. Quem sabe alguém encontra aqui algo interessante!

2 - O PIC10F200

O PIC10F200 é o microcontrolador mais simples que encontrei. Ele conta com uma memória RAM de apenas 16 bytes e uma ROM de 384 bytes, que pode conter 255 instruções. A última posição da memória ROM contém o valor de calibração do oscilador interno do PIC e não pode ser usada. O diagrama com os pinos do PIC10F200 é mostrado abaixo.



Antes que você estranhe, melhor eu dizer. O programa fica gravado numa memória Flash. O PIC10F200 tem 384 bytes de memória Flash. Mas eu sou da velha guarda, quando esse tipo de memória era chamada de ROM, então eu vou chamar de ROM mesmo.

O PIC10F200 conta com quatro pinos de entrada/saída, GP0, GP1, GP2 e GP3. GP3 é um pino de entrada apenas. Os demais podem ser configurados como entrada ou saída. A configuração de GP0 e GP1 é a mesma de qualquer outro PIC, bastando zerar os bits 0 e 1 do registrador TRIS, mas GP2 e GP3 tem algumas particularidades.

O GP3 pode operar como entrada ou como reset do PIC. Para que opere como entrada, é necessário colocar a diretiva *#pragma config MCLRE = OFF* no início do código.

O GP2 acumula várias funções, podendo operar como entrada, saída, saída de clock ou

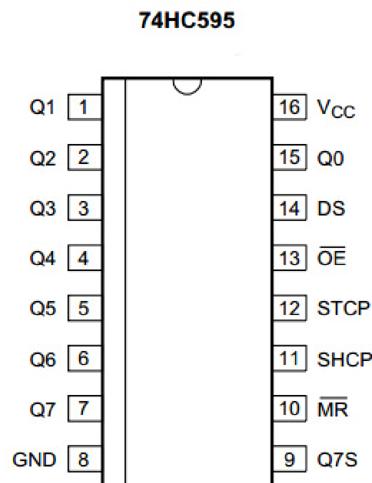
entrada de clock para o timer interno. Para que funcione como saída, é preciso desabilitar a entrada de clock, zerando o bit 5 do registrador OPTION antes de zerar o bit 2 do registrador TRIS.

O pino 2 do PIC é ligado a alimentação e o 7 ao aterramento. Os pinos 1 e 6 não possuem conexão interna.

É importante que a tensão no pino 8 (GP3) nunca seja de 12V ou mais, ou o PIC entrará em modo de programação.

3 - Multiplicando as saídas

Em termos de comunicação paralela três saídas são muito pouco para quase qualquer coisa que se queira fazer. Entretanto, comunicação serial requer apenas duas saídas! E existem circuitos integrados baratos que fazem justamente isso. Um deles é o 74HC595, que é um registrador de deslocamento (*shift register*). A pinagem do integrado é mostrada na figura abaixo. O integrado opera respondendo aos níveis lógicos presentes nos seus pinos de entrada. O nível lógico 1 corresponde a ligar um pino à tensão de alimentação VCC e o nível lógico 0 corresponde ao aterramento.



3.1 - O registrador 74HC595

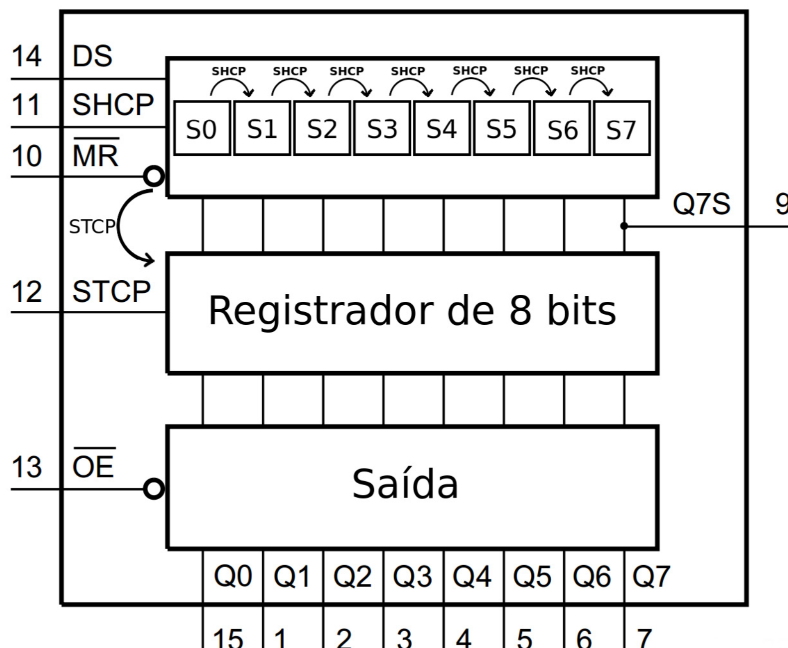
O funcionamento do integrado 74HC595 está esquematizado na próxima imagem. Cada vez que o sinal SHCP, aplicado no pino 11, passa de 0 para o valor lógico 1, o conteúdo do registrador S7 é perdido e substituído pelo dado presente em S6. Por sua vez, S6 perde seu valor e passa a conter o valor de S5 e assim sucessivamente até S0, que perde seu dado e passa a conter o valor de DS, pino 14.

Imagine que, por exemplo, em um dado momento todos os registradores contêm o valor lógico 0. Deixemos o pino 11 (SHCP) aterrado e vamos ligar o pino 14 (DS) a VCC. Neste momento o pino 14 está no nível lógico 1. Então tiramos 11 do aterramento e ligamos a VCC e voltamos a aterrá-lo. No instante que 11 passa de 0 para o nível lógico 1, S0 passa a conter o valor lógico 1 presente no pino 14 e os demais registradores permanecem zerados. Ao voltar 11 para 0, nada acontece.

Se agora aterrarmos o pino 14 e novamente levamos 11 para o nível lógico 1, o dado de S0 é transferido para S1 e S0 passa a conter 0, que é o dado presente em 14. A cada vez que o pino 11 é levado para o nível lógico 1, o bit anda um registrador para frente. Desta forma, alternando-se coordenadamente os níveis lógicos dos pinos 11 e 14, é possível colocar qualquer sequência de dados nos registradores S0 a S7.

Quando o pino 12 é levado para o nível lógico 1, o valor dos registradores SX são copiados para o registrador de 8 bits e reproduzidos nas saídas Q0 a Q7, enquanto o pino 13

estiver no nível lógico 0. Se 13 vai para 1, as saídas são zeradas, embora nenhum registrador seja alterado.



Desta forma o 74HC595 transforma uma entrada de dados seriais em uma saída paralela, juntando um punhado de bits para fazer um byte.

Há mais dois pinos importantes para discutir. O pino 10 é o reset mestre do registrador de deslocamento. Quando é levado para o nível 0, todos os registradores S passam a conter 0. Para que o integrado funcione, ele deve estar no nível 1.

O pino 9 é o que torna o 74HC595 realmente útil. Ele contem o valor do registrador S7. Se ligarmos o pino 9 de um 74HS595 ao pino 14 de outro, e juntarmos os pinos 11 e 12 de um com os correspondentes do outro, cria-se um registrador de 16 bits!

Na verdade não há limite para quantos 74HC595 são colocados em sequência. Pode-se conseguir qualquer quantidade de saídas que se queira, entretanto, para sequências muito longas, o tempo de preenchimento dos registradores pode ser um problema.

4 - Números binários

Já faz parte do conhecimento popular que computadores, calculadoras, telefones celulares, microcontroladores e tudo o que é digital, trabalham em base binária. Entretanto pouca gente entende de fato o que é a base binária e como ela funciona.

4.1 - Base binária

Nos dispositivos digitais, qualquer dado é armazenado na forma de zeros e uns, que são os bits. Uma imagem, um som ou um texto, são convertidos em números e armazenados na forma de bits. Para facilitar a operação, os bits são organizados em grupos de oito, formado bytes. O número binário 10010010 é um byte. Os bits são numerados de 0 a 7 da direita para a esquerda, sendo o mais a esquerda é chamado de mais significativo e o a direita de menos significativo. Para converter um byte em um número decimal, basta multiplicar cada bit pela potência de 2 corresponde a sua posição e somar os valores, da seguinte forma:

$$1\cdot 2^7+0\cdot 2^6+0\cdot 2^5+1\cdot 2^4+0\cdot 2^3+0\cdot 2^2+1\cdot 2^1+0\cdot 2^0=128+0+0+16+0+0+2+0=146$$

Portanto, 10010010 é a forma binária do número 146. Com um byte é possível escrever qualquer número entre 0 e 255, sendo zero escrito como 00000000 e 255 como 11111111.

É fácil perceber que a conversão da base binária para a decimal requer algum esforço, mas não é algo que traga grandes problemas. Fica realmente complicado quando se pretende trabalhar com números fracionários. Frações em binário são expressas na forma somatório de potências negativas de 2. Assim, em binário, uma fração é representada como 1/2+1/4+1/8 e assim por diante. O problema é que números como 0,2 (dois décimos) não podem ser escritos de forma exata usando potências de 2. Em base binária, 0,2 é uma dizima, assim como 1/3 é uma dizima em base decimal.

Por isso que quando computadores são programados para fazer cálculos que exigem grande precisão, são usados 32, 64 ou até 128 bits para representar cada número. Mas calculadoras de bolso, microcontroladores e mesmo computadores antigos não dispõem de potencia computacional suficiente para processar tantos bits. Por isso, alguns dispositivos usam Decimais Codificados em Binário, ou BCD, do inglês *Binary Coded Decimals*.

4.2-Decimais codificados em binário (BCD)

No sistema BCD, o byte é tratado não mais como um número em si, mas como uma codificação para dois dígitos decimais. Assim, os quatro bits menos significativos representam as unidades e os quatro mais significativos, as dezenas. Em BCD, o nosso byte 10010010 seria traduzido como:

$$1\cdot80+\ 0\cdot40+\ 0\cdot20+\ 1\cdot10+\ 0\cdot8+\ 0\cdot4+\ 1\cdot2+\ 0\cdot1=92$$

A mesma representação binária que antes significava 146, agora, na convenção BCD representa 92. A amplitude dos valores que se pode codificar cai bastante, pois agora temos apenas números de 0 a 99 e estamos gastando o mesmo espaço de memória. Por outro lado, o problema da dízima periódica binária para as frações decimais desaparece. O gasto maior de memória é tolerado principalmente em se tratando de somas monetárias. Se os centavos viram dízimas periódicas, imagine o prejuízo dos bancos.

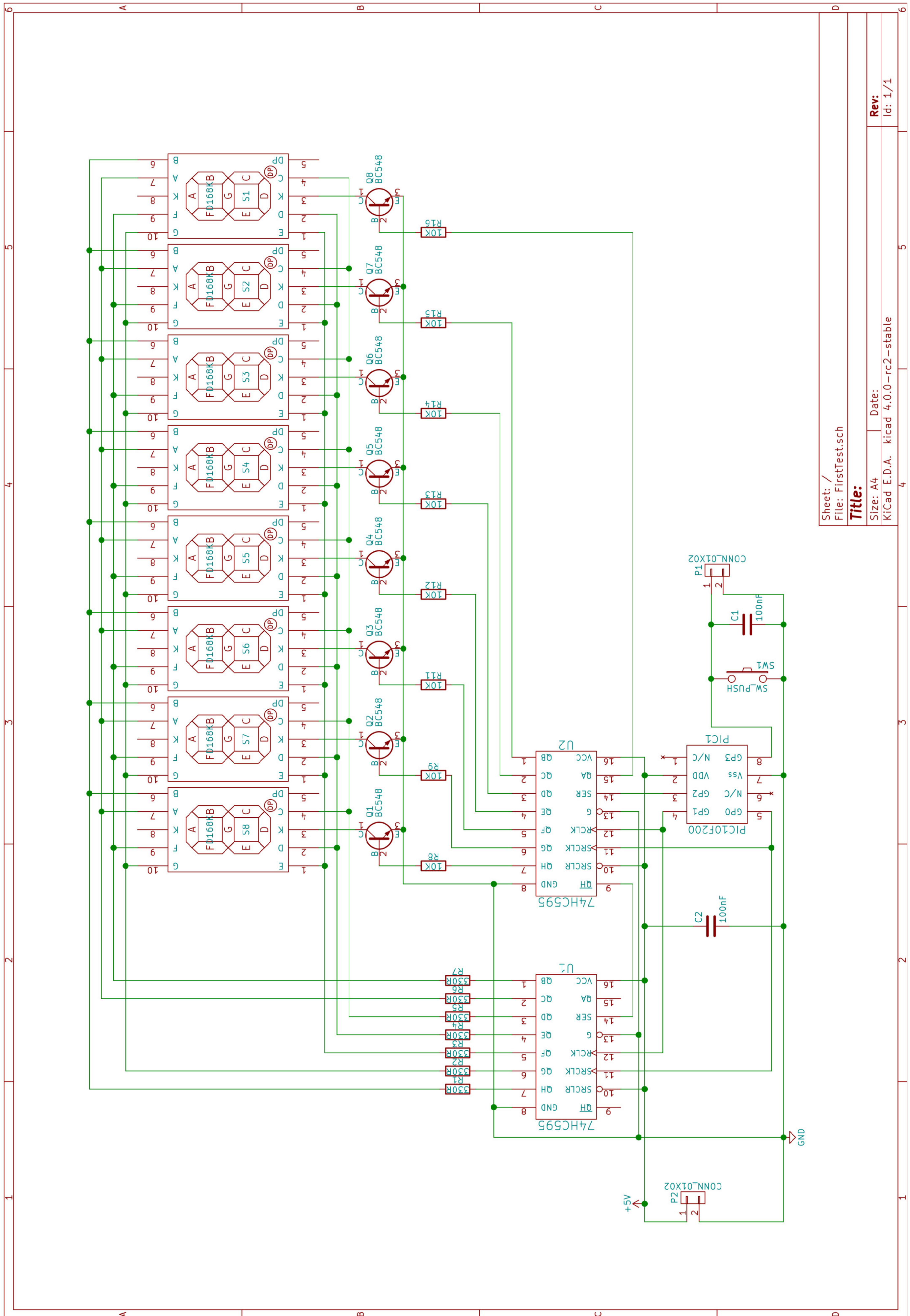
Fazer contas com BCD em um processador binário nem sempre é fácil. Os antigos Z80 e 8080 tinham instruções próprias para tratar BCD, mas não é o caso dos microcontroladores. Os algoritmos de aritmética BCD precisam ser implementados manualmente. No entanto, esse método é muito mais simples que fazer as contas em binário puro e depois traduzir para base decimal antes de apresentar o resultado no display do contador. Por isso, neste projeto, a contagem é feita em BCD.

5 - O circuito

É bem fácil construir um contador como este usando apenas o integrado CD4026 ou o CD4033. Ambos são contadores até 10 com saída para display de sete segmentos. Oito deles ligados corretamente e pronto. Entretanto, criar uma placa de circuito impresso para um projeto assim será um belo desafio. Cada pino dos integrados tem sua função definida e não pode ser alterada.

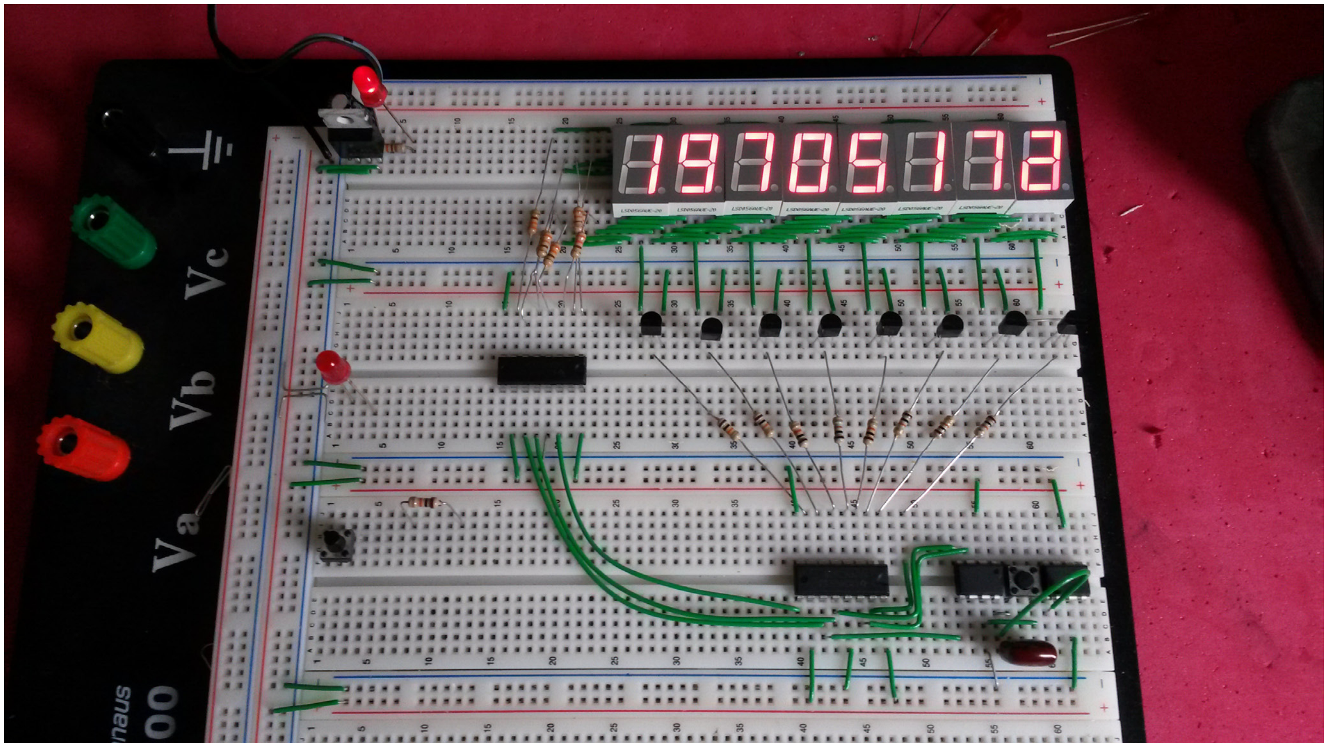
A vantagem em se usar microcontroladores em vez de circuitos integrados específicos é a liberdade ganha. Uma porta do microcontrolador pode ser qualquer coisa. O autor do projeto decide o que liga onde.

O circuito do contador, visto na próxima página, é fruto de muito retrabalho. Em virtude do uso do microcontrolador, que, como dito, flexibiliza a alteração do uso dos seus terminais, sempre que uma trilha na placa de circuito ficava complicada, nada impediu alterações para facilitar o traçado. Isso foi feito diversas vezes.


Sheet: /
File: FirstTest.sch
Title:
Size: A4
Date:
Rev:
KiCad E.D.A. kicad 4.0.0-rc2-stable
Id: 1/1
74HC595
U1
U2
PIC10F200
PIC1
BC548
FD168KB
10K
330R
C1
C2
100nF
SW1
SW_PUSH
P1
P2
CONN_01X02
GND
+5V

Acho importante dizer isso, pois o HandMaker iniciante, que pega um documento como este, tem a ilusão que os autores são algum tipo de bruxo. Seres transcendentais que conseguem imaginar coisa complexas que dão certo de primeira. Não é assim. Montei o protótipo em uma protoboard, vista abaixo, e alterei as ligações inúmeras vezes até funcionar. E, mesmo depois de pronto e montado em sua placa, identifiquei mais um problema. Felizmente não perdi a placa. Bastou um remendo.

O recado que quero passar com essa lenga-lenga é que você não deve desanimar se seu projeto não dá certo de primeira, ou de segunda, ou de décima oitava. Projetos HandMades são fruto de muita lapidação e nem tudo o que fazemos dá certo.



Voltando ao assunto, os integrados U1 e U2 são responsáveis por fazer os números aparecerem no display. U1 determina quais segmentos vão acender e U2 qual display será ativado. Sua ligação forma um registrador de 16 bits. Assim, o microcontrolador informa nos primeiros 7 bits quais segmentos são ligados. Daí envia um 0 para completar os 8 primeiros e envia os bits que determinam o display a ser mostrado. Apenas um display fica aceso a cada vez. Isso é o que se chama de display multiplexado. Fazendo isso rápido o suficiente, cria-se a ilusão que todos estão acesos ao mesmo tempo.

Isso é feito usando os pinos 11, 12 e 14 de U1 e U2 e as portas GP0 GP1 e GP2 do PIC10F200. GP1 fornece o clock, entanto GP2 envia os dados na sequência. O pino 14 de U1 é ligado ao pino 9 de U2 para que os dados possam continuar o caminho.

Uma vez enviados todos os 16 bits necessários, GP0 é levado ao nível lógico 1 e trazido para 0 novamente, jogando os dados para o display.

U2 terá apenas uma das saídas em nível 1, ativando apenas um display por meio de um dos transistores BC548. Já U1 terá várias saídas em nível 1, ativando os segmentos adequados.

A porta GP3 do PIC é a entrada do contador. O PIC é configurado para que GP3 tenha sempre o nível lógico 1 caso não esteja conectado. Quando a chave SW1 é acionada, GP3 vai para zero e o contador recebe um incremento.

O capacitor C1 serve para filtrar o ruído gerado pela chave. Sem ele, cada vez que a chave é acionada, a contagem pode pular vários dígitos cada vez.

Pode-se também contar eventos externos, bastando ligar uma outra chave no conector P1.

O capacitor C2 estabiliza a tensão para o PIC e precisa ser montado na placa perto do microcontrolador.

O circuito é alimentado com 5V através do conector P2. Também pode ser alimentado com 3 pilhas AAA, fazendo 4,5V, já que todos os componentes funcionam adequadamente com tensões entre 3V e 5,5V.

6 - O programa

Esta é a parte deste documento que, provavelmente, você terá mais dificuldade de entender. Não se frustre. Se ficar confuso, vá tomar um café e volte a ler novamente. Cafeína costuma ajudar nessas coisas. Programação é uma mistura de ciência e arte que faz mesmo os iniciados arrancarem os cabelos. Programar um microcontrolador como o PIC10F200 é um desafio. Estamos acostumados a tratar computadores que tem vários gigabites de memória e operam com giga Hertz de clock. O PIC10F200 tem apenas 16 bytes de memória RAM e 384 bytes de ROM, onde fica o programa e um clock de 4MHz. É fácil pensar que nada de útil pode ser feito com tão pouco recurso. Mas devemos lembrar que na década de 1950 e começo dos anos 1960, grandes empresas pagavam fortunas por computadores com recursos desta mesma ordem de grandeza. Hoje compramos com moedas.

O programa foi escrito em linguagem C usando o MPLab X e o compilador XC8 da Microchip. Escolhi esta plataforma de trabalho por várias razões. Primeiramente é um software que pode ser baixado livremente no site do fabricante dos microcontroladores. Segundo, eu queria mostrar que não é obrigatório o uso de linguagem assembler para programar microcontroladores pequenos. Basta saber como fazer e tomar certos cuidados.

Grande parte das dicas para se gerar um programa curto são dadas no próprio manual do XC8. Por exemplo, comparações de igualdade ou desigualdade geram códigos de máquina mais curtos que comparações de maior ou menor. Da mesma forma, comparações com 0 ou 1 geram código mais curto que com outros números. Por esta razão, procurei fazer os laços FOR com contadores decrescentes parando em 0. Isso economizou vários bytes de ROM

Também é mais curto comparar coisas do mesmo tipo. Por exemplo, o código para comparar dois bits, ou dois bytes, é mais curto que o gerado para comparar um bit com um byte.

6.1 - Declarações

O programa completo pode ser visto na listagem que está no final do documento. Na linha 10 o compilador é instruído a desabilitar o temporizador interno do PIC10F200 (*Watch Dog Timer*). Caso isso não seja feito, o microcontrolador irá reiniciar a intervalos regulares, perdendo a contagem. Na linha 11 é desativada a proteção do código. Como este é um projeto livre, não há sentido em habilitar a proteção. A linha 12 é importante. Aqui o pino 8 do PIC (GP3) é configurado como entrada. Sem esta linha ele opera como reset.

A linha 14 é apenas um formalismo. Ela informa ao compilador qual a frequência de clock do PIC10F200. Esta informação não será usada neste projeto, mas é bom ter o costume de incluir este tipo de informação.

Das linhas 16 a 20 são definidos apelidos mnemônicos para GP0 a GP3. GP0 será *ClockOut*, GP1 *ShowData*, GP2 *DataOut* e GP3 *DataIn*.

Nas linhas 24 a 33 são definidos os bits que correspondem aos segmentos do display. Note que os segmentos estão embaralhados. Seria esperado que fossem codificados como ABCDEFG, mas para facilitar o desenho da placa de circuito, as letras foram reordenadas. O bit menos significativo corresponde ao ponto decimal, que não é conectado. Por isso é sempre enviado um zero. Na verdade não faz diferença se enviamos 0 ou 1, pois a conexão física foi omitida no circuito, mas algo deve ser enviado para completar o preenchimento dos registradores de deslocamento.

Temos então as declarações das funções e das variáveis globais. A contagem é acumulada na matriz *Counter[]*, linha 38. Ela consiste de 4 bytes que irão conter a contagem em BCD.

A variável *NonZero*, linha 39, serve para regular a varredura do display. Não há sentido em mostrar zero em todas as posições a esquerda de um número. Não só torna a leitura desconfortável, como também consome energia. Vou explicar isso melhor mais adiante, quando discutirmos a função **SendData()**.

6.2 - Função main()

O programa começa na função **main()** que inicia-se por configurar o microcontrolador. Por padrão todos os bits do registrador OPTION são 1 quando o PIC10F200 é ativado, Caso essa configuração seja adequada, nada precisa ser feito. Mas neste projeto, temos que mudar algumas coisas. Na linha 45 são zerados os bits 5 e 6 do registrador OPTION (lembre que os bits são numerados de 0 a 7).

O bit 5, quando configurado em 0, indica que o pino 3 do PIC10F200 (GP2) irá operar como a entrada/saída e, quando configurado em 1, como entrada de clock para o temporizador interno. Como queremos uma saída, o bit deve ser zerado.

O bit 6, quando 0, ativa o pull-up das entradas GP0, GP1 e GP3. Isso quer dizer que, a menos que exista algo forçando a entrada a zero, ela será considerada 1. Como neste projeto a contagem é feita ligando o pino 8 (GP3) ao terra, é preciso algo que o leve de volta para 1 quando a conexão for desfeita. O mesmo efeito pode ser obtido ligando um resistor de 10k entre o pino 8 e o VCC, e esta foi minha primeira opção. Entretanto achei mais interessante mudar o bit e tirar um resistor da placa. O HandMaker atendo sentirá a falta de R10 no projeto. Tirei o resistor, mas não mudei a numeração.

O registrador TRIS define se um determinado pino é entrada ou saída. Na linha 45 o pino GP3 é definido como entrada, e os demais como saída. Na verdade, GP3 é sempre entrada, não importa o que exista no registrador TRIS, portanto bastaria apenas zerar os 3 bits menos significativos. Preferi escrever a linha completa por disciplina, uma vez que se trata de um projeto didático.

Nas linhas 46 e 47 temos a declaração das variáveis *Vrd*, que faz o controle da varredura de multiplexação dos displays, e *GPM*, que guarda o último valor de GP3 para controle da contagem.

Começa então um laço WHILE na linha 48. Este laço tem como condição um parâmetro que é sempre verdadeiro, portanto o programa nunca termina. Faz sentido! Queremos contar sempre, e não até um certo ponto. Cada vez que o laço é iniciado, a variável *NonZero* é zerada, desligando os displays. Este desligamento terá efeito sobre a função **SendData()**.

O laço FOR que começa na linha 50 é o controle principal da varredura dos displays. Os dígitos são apresentados da esquerda para a direita, por isso a contagem vai de 8 a 1 em ordem decrescente. Na linha 51 é chamada a função **SendData()**, explicada mais adiante, que envia o dígito *Vrd* para o display correspondente. O importante a saber agora é que quando **SendData()** termina, todas as saídas estão zeradas.

Na linha 42 o valor de *GPM* é comparado com o registador GPIO. Na primeira versão deste programa, eu comparava *GPM* com *DataIn* (GP3), mas como *GPM* é um byte e *DataIn* um bit, o código de máquina para esta comparação ficava um pouco longo. Comparando *GPM* com o registrador completo gera um código menor. Mas para não ter problema nessa linha, é importante que todos os demais bits de GPIO sejam zero.

Se GPIO é diferente de GPM, então houve uma mudança no valor de *DataIn*. Na linha 53 verifica-se se *DataIn* é zero. Se for, então a tecla foi acionada. A primeira coisa a fazer é zerar GPM, para que seja novamente igual a GPIO, indicando que este acionamento já foi contado. Então é chamada a função **Increment()**, que incrementa o contador.

Se *DataIn* não for zero, significa que a chave foi liberada e o programa passa pela linha 57, fazendo GPM ter o valor binário 0b1000, que corresponde a GPIO com o valor 1 em *DataIn*. Com isso o laço FOR se encerra, passando para a varredura do próximo display ou, se *Vrd* já chegou em 1, reiniciando o laço WHILE para a próxima varredura.

6.3 - A função SendData()

A função **SendData()** é responsável pelo envio dos dados aos displays. O parâmetro *Ind* representa qual display deve ser mostrado. Como a rotina é chamada de um laço FOR decrescente na função *main()*, a varredura ocorre da esquerda para a direita.

Os números a serem apresentados estão contidos na matriz *Counter[]* em BCD. Portanto a primeira coisa a fazer é traduzir o parâmetro *Ind*, que varia de 1 a 8, para o índice da matriz *Counter[]*, que varia de 0 a 3. Isso é feito nas linhas 77 e 78. Primeiro, subtrai-se 1 do valor de *Ind*, obtendo uma variação de 0 a 7. Este valor é então dividido por 2 com arredondamento para baixo, chegando no índice de *Counter[]* que contém o dígito procurado. Esta divisão, em binário, corresponde a rolar todos os bits de um número uma posição para a direita (linha 78). Dividir números binários por 2, 4, 8, etc, é tão fácil quanto dividir números decimais por 10, 100, 1000, etc. Nos dois casos, basta rolar o número para a direita.

O valor de *Counter[]* é então copiado para a variável *Sending*. Como cada posição em *Counter[]* contém dois dígitos, é preciso saber se devemos mostrar a dezena ou a unidade. Para isso, basta verificar se *Ind* é par ou impar. Os valores ímpares de *Ind* correspondem a unidades em *Counter[]*; os pares, dezenas.

Em binário, todo número par terminam em 0, e ímpar em 1. Logo, para saber se *Ind* é par ou ímpar, basta verificar o bit menos significativo. Na linha 81 é feita essa verificação, zerando todos os bits, exceto o menos significativo. Caso *Ind* seja ímpar, então os bits referentes a dezena são zerados em *Sending* na linha 81. Se for par, os bits da dezena são rolados para a esquerda, passando a ocupar a posição das unidades. Nos dois casos, terminamos apenas com os quatro bits menos significativos.

É muito mais confortável ler 3409 nos displays que 00003409. Por isso, é desejável ocultar os zeros a esquerda. Como a varredura ocorre da esquerda para a direita, é fácil fazer isso. Basta saber se um dígito diferente de zero já foi mostrado. Dessa forma, os quatro primeiros zeros do exemplo não aparecerem, mas o da posição 2 sim, pois há dígitos diferentes à sua esquerda. Mas se todos os dígitos são zero, como é o caso quando o contador é ligado, um zero deve aparecer na primeira posição, para indicar que a contagem pode começar.

Tudo isso é feito na linha 86. Se *Sending* é diferente de zero ou se *Ind* é 1, *NonZero* recebe o valor 1. Então, uma vez encontrado um dígito não nulo, todos os dígitos a partir daquele serão mostrados, já que *NonZero* só será anulada novamente na função *main()*, antes do início da próxima varredura.

O HandMaker atento irá pensar consigo: “Ora! Se *NonZero* é zero, então a rotina poderia ser interrompida nesse ponto! Não faz sentido continuar se no final o display ficará apagado!”. O pensamento está correto. Entretanto, se a rotina for interrompida, a frequência da multiplexação irá variar conforme o número de dígitos apresentados. Isso causará uma variação no brilho dos displays conforme a contagem for progredindo. Para evitar este efeito, é necessário inserir um atraso na execução cada vez que *NonZero* for zero. A forma mais fácil de criar este atraso é deixar a rotina seguir seu fluxo.

Nas linhas de 89 a 98 o valor em *Sending* é comparando e trocado pela sequência de bits necessária para criar o número no display de sete segmentos. Não é feita a comparação com 9, já que ela pode ser assumida por exclusão.

O aficionado por programação em C pode questionar porque não usei um SWITCH – CASE em lugar do IF – ELSE IF. Neste caso, as duas construções são perfeitamente equivalentes, entretanto esta gerou um código de máquina um byte mais curto.

O laço FOR iniciado na linha 100 envia os 8 bits referentes aos segmentos que devem acender no display. A contagem é decrescente para economizar uns bytes de código de máquina pela comparação com zero. É assumido que o laço começa com todos os bits de entrada de GPIO zerados. Na linha 101 o bit mais significativo de *Sending* é verificado. Se for diferente de zero, então *DataOut* é feito 1. Verificar se o bit é igual a 1, do ponto de vista lógico dá o mesmo resultado. Porém, mais uma vez, comparar com zero produz um código de máquina dois bytes mais curto.

Então, na linha 104, *ClockOut* é levado a 1, armazenando o bit nos registradores 74HC595. Em seguida, tanto *ClockOut* quanto *DataOut* são zerados na linha 105. Na verdade todo o registrador GPIO é zerado. Zerar o registrador todo é mais rápido e produz um código mais curto que zerar os dois bits separadamente. Na linha 106 é feito um deslocamento à esquerda de *Sending*, o que coloca o segundo bit mais significativo na posição do primeiro, deixando tudo pronto para o próximo envio.

O laço FOR iniciado na linha 108 irá enviar o bit que define qual display será ativado. Isso é feito colocando 1 em *DataOut* quando a variável contadora *i* for igual ao parâmetro *Ind*, contanto que *NonZero* seja diferente de zero. Se *NonZero* for zero, nenhum bit será enviado e o display corresponde ficará apagado. Novamente, *ClockOut* é levado a 1 e em seguida zerado o registrador GPIO.

Terminado o segundo laço FOR, *ShowData* é alternado para 1 e novamente para 0, fazendo os dados dos registadores de deslocamento migrarem para as saídas. É importante notar que a função termina com GPIO zerado. Caso isso não ocorra, haverá problema na contagem na função *main()*.

6.4 - A função Increment()

Esta é uma função curta, mas de grande importância neste projeto. Não só é onde a contagem realmente ocorre, mas também pela forma como é feita. Como expliquei no item 4.2, a operação é feita em BCD, enquanto o microcontrolador trabalha em base binária. O que esta rotina faz é controlar aquele “vai um” das somas.

Inicialmente são declaradas algumas variáveis para controle e operação da função. A primeira, na linha 126, é *Carry*, que contém o “vai um”. A variável *i* é um contador que determina qual posição de *Counter[]* será operada. *WorkVal* irá conter uma cópia local do valor de *Counter[]*.

Na linha 129 começa o laço WHILE que irá perdurar enquanto *Carry* for diferente de zero, ou seja, há algo a ser somado, e *i* for diferente de 4, significando que ainda não chegou o final da matriz *Counter[]*.

Começa-se por copiar o valor de *Counter[]* para *WorkVal*, na linha 132. É possível fazer toda a rotina funcionar sem usar *WorkVal*, acessando diretamente *Counter[]*. Mas esta abordagem gera um código de máquina mais longo, devido às instruções extras necessárias para acessar os valores na matriz. Como eu disse no início, programar o PIC10F200 é um desafio. A RAM é pequena e a ROM também. Em alguns casos, como aqui, gasta-se uma para economizar a outra.

A linha 137 merece uma atenção especial. A explicação dela vai fazer sentido para quem já programou em assembler e tem conhecimento de arquitetura de processadores. Se este não é o seu caso e você se sentir perdido, apenas siga em frente. A coisa melhora mais adiante.

O que a linha 137 faz é verificar se os quatro bits menos significativos de *WorkVal* são diferentes de 9. Originalmente eu havia escrito esta linha como:

if ((WorkVal & 0b00001111)!=9) {…..}

É a forma mais óbvia e direta de fazer a comparação. Mas quando verifiquei o código assembler gerado pelo compilador, notei que ele estava gastando um byte de memória para guardar o valor de *(WorkVal & 0b00001111)* para então fazer uma operação XOR com 9 e daí comparar com zero. A mesma coisa pode ser feita colocando *WorkVal* no registrador W do PIC10F200, fazer um XOR com 9 e depois um AND com 0B00001111, antes de comparar com zero. Então mudei a linha para a forma como está na listagem:

if (((WorkVal ^ 9) & 0b00001111)!=0){...}

Esta forma não é óbvia, não é clara, não é bonita, não é intuitiva e parece obra de um lunático, mas economiza um byte de RAM e seis de ROM.

Bom, voltemos ao planeta terra. Digamos que os quatro bits menos significativos de *WorkVal* não sejam 9. Neste caso, soma-se 1 a *WorkVal* na linha 138 e zera-se *Carry* em seguida. Fim de conversa! A função termina e seguimos felizes!

Mas e se for 9? Ora, lembre da Tia Ermengarda, do segundo ano do primário, nove mais um dá dez, “fica zero e vai um”! *Carry* já é um. Falta o “fica zero”. É isso que é feito na linha 141, depois do ELSE. É feita uma operação AND binária com *WorkVal* e 0b11110000. Isso vai zerar os quatro bits menos significativos, preservando os demais. Agora é só somar o *Carry*, contanto que os quatro bits mais significativos sejam diferentes de 9.

Chegamos então na linha 142. Esta é simples e direta. Se não for 9, soma 1 e zera o *Carry*. Mas perceba que a comparação é feita com 0x90, porque zeramos os quatro bits menos significativos e precisamos comparar a dezena. A soma na linha 143 também pode parecer estranha, mas significa somar 1 na posição referente à dezena.

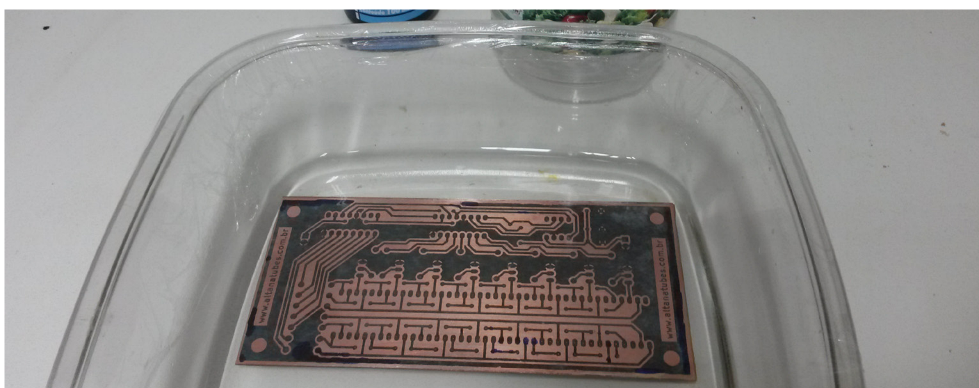
Se, por outro lado, *WorkVal* tiver um 9 na posição da dezena, então novamente temos o “fica zero e vai um” e caímos na linha 147. *Carry* ainda é um e precisamos zerar as dezenas agora. Mas para chegar na linha 147, as unidades foram zeradas antes. Logo, *WorkVal* vai conter zero em todas as posições. Nenhuma operação especial é necessária. É só zerar tudo.

Na linha 150 o resultado de *WorkVal* é colocado de volta na matriz e *i* é incrementada para continuar o processo, caso ainda haja 1 em *Carry*.

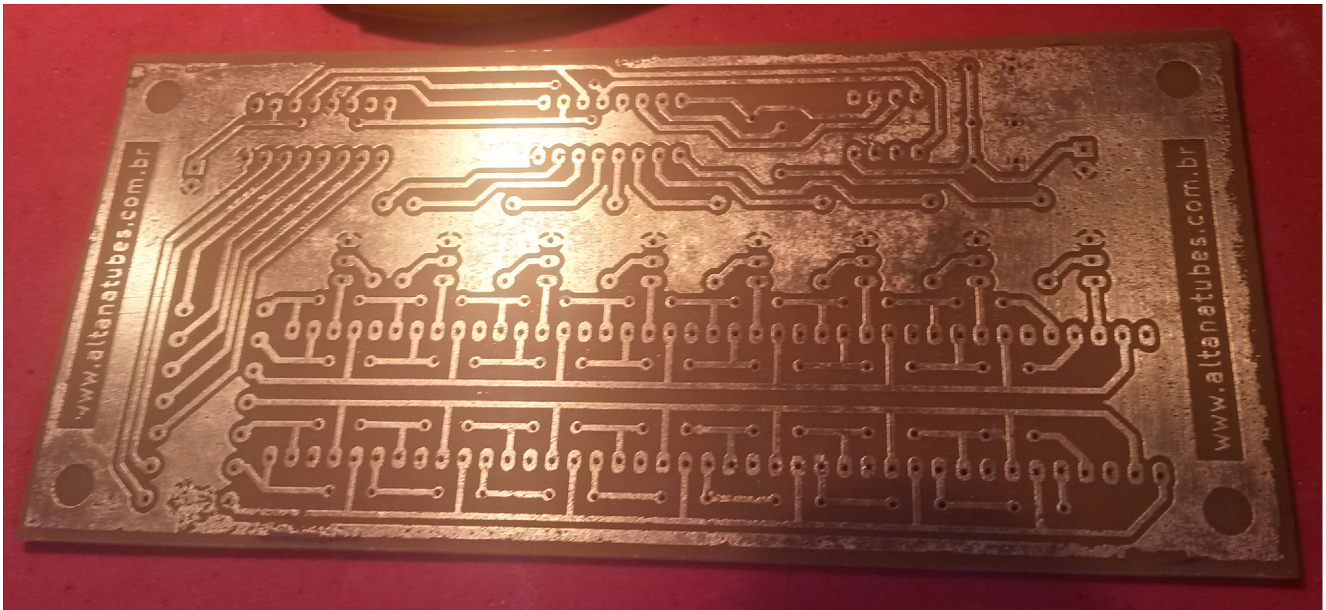
Você agora deve estar pensando que, passado o trauma inicial da linha 137, o resto da explicação parece bem boba. E é mesmo. Operar com números em BCD é ensinar a máquina a fazer contas como nós aprendemos na escola quando tínhamos 10 anos de idade. Fora toda a sofisticação aparente, com operações lógicas de nomes estranhos, tudo o que esta função faz é reproduzir a lógica da adição de dois números em base decimal.

7 - O protótipo.

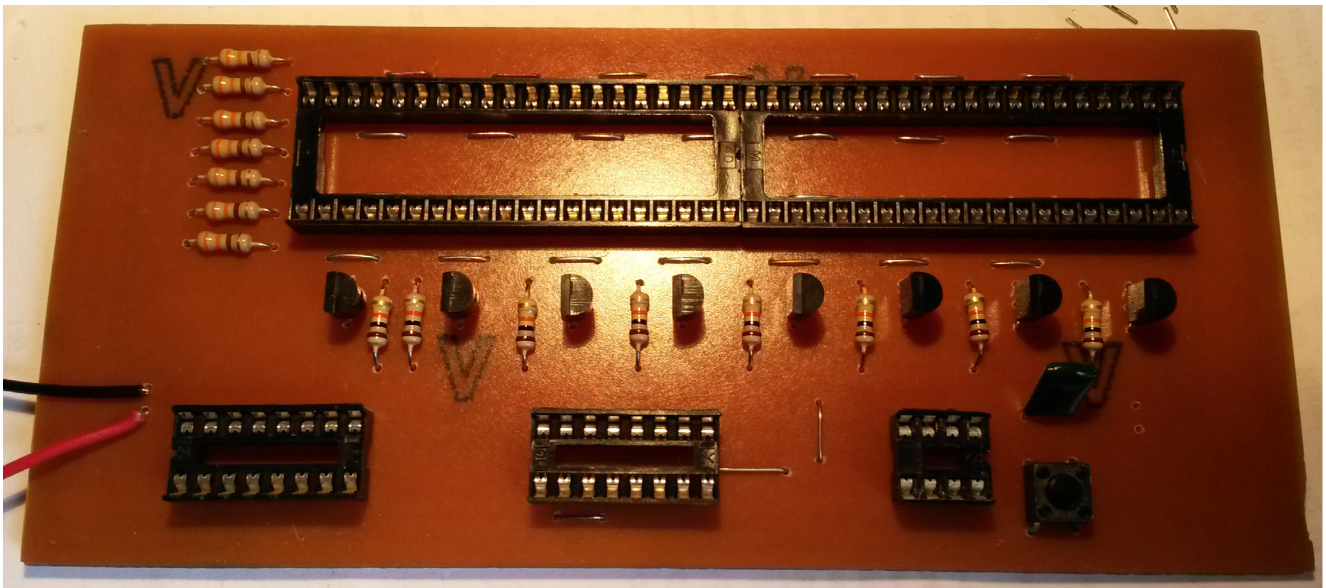
Aqui eu apenas quero, *mui* orgulhosamente, mostrar a coisa montada. Na próxima imagem pode ser vista a placa, que foi feita por transferência de toner e corroída com vinagre e água oxigenada seguindo a receita do Fórum HandMades.



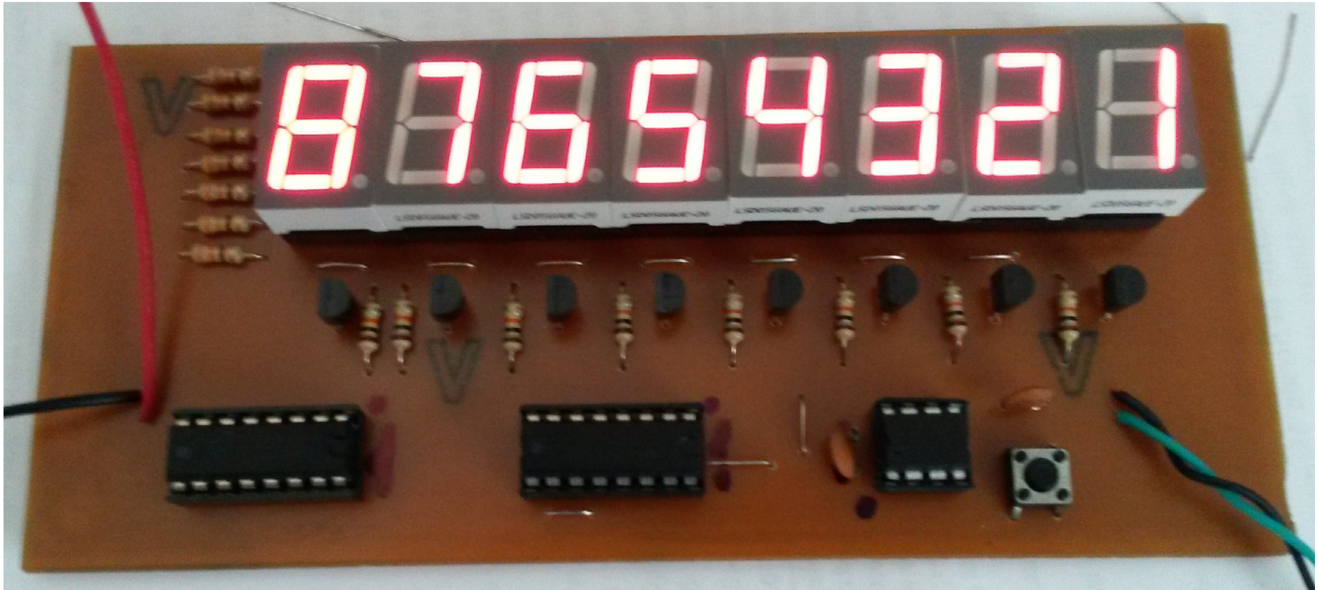
O resultado final está na Figura 6. Houve umas falhas na transferência, mas nada grave.



Para manter o costume dos montadores de valvulados, coloquei soquetes em tudo, até nos displays, como visto na Figura 7.



A coisa pronta e funcionando é mostrada na próxima página. Tenho que confessar uma trapaça: não apertei a tecla 87.654.321 vezes. Gravei um PIC10F200 sabotado para começar nesse número.



8 - Conclusões

A primeira e mais importante conclusão que se pode tirar este projeto é não desprezar um microcontrolador por ser pequeno. Eles são capazes de fazer coisas bastante sofisticadas. A maioria dos problemas pode ser resolvido com algo pequeno. Antes de partir para algo com mil terminais, pare e pense.

Em segundo lugar, ficou claro para mim que o circuito, o programa e o layout da placa devem ser desenvolvidos ao mesmo tempo. A possibilidade de uma alteração pequena em um deles proporcionar facilidades nos outros é muito grande para ser desprezada.

9 - Listagem do programa

```
1.  /*
2.   * File:   main.c
3.   * Author: Eduardo Foltran
4.   *         www.altanatubes.com.br
5.   * Created on 14 de Novembro de 2015, 07:26
6.   */
7.
8.  #include <xc.h>
9.  // CONFIG
10. #pragma config WDTE = OFF    // Watchdog Timer (WDT disabled)
11. #pragma config CP = OFF      // Code Protect (Code protection off)
12. #pragma config MCLRE = OFF   // Master Clear Enable (GP3/MCLR pin fuction is
13.                              // digital I/O, MCLR internally tied to VDD)
14. #define _XTAL_FREQ 4000000 // Define a frequencia do clock interno do PIC
15.
16. #define ClockOut GP0         // GP0 conterá o sinal de clock para o 74HC595
17. #define ShowData GP1         // Ativa o display para os dados transmitidos
18. #define DataOut  GP2         // A porta GP2 conecta com a entrada (14) do
19.                              // primeiro 74HC595
20. #define DataIn   GP3         // A entrada do contador será feita por GP3
21.
```

```
22. // Definição do display:
23. // Segmento       BGEDCAFP        A codificação do display de 7 segmentos
24. #define Number0  0b10111110      // foi arbitrada para facilitar o desenho
25. #define Number1  0b10001000      // da placa de circuito
26. #define Number2  0b11110100
27. #define Number3  0b11011100
28. #define Number4  0b11001010
29. #define Number5  0b01011110
30. #define Number6  0b01111110
31. #define Number7  0b10001100
32. #define Number8  0b11111110
33. #define Number9  0b11011110
34.
35. void Increment(void);
36. void SendData(char Ind);
37.
38. char Counter[4]={};          // Matriz com os digitos do contador
39. char NonZero;                // Passa a ser 1 quando a varredura encontra
40.                              // um dígito diferente de zero
41.
42. void main(void) {
43.    OPTION=0b10011111;           // Bit 5 é zero para habilitar GP2 como saída
44.                                 // Bit 6 é zero para habilitar pull up em GP3
45.    TRIS=0b1000;                 // Define todos os pinos como saída exeto GP3
46.    char Vrd;                    // Vrd é o controle da varredura
47.    char GPM = 0;                // Guarda o valor de GP3 para detectar a transição
48.    while (1) {                  // Aqui fica o loop principal do programa
49.       NonZero=0;                // Desliga a impressão de zeros a esquerda
50.       for (Vrd=8;Vrd!=0;Vrd--){ // A varredura é feita da esquerda pra direta
51.          SendData(Vrd);         // Envia para o display o dígito da vez
52.          if (GPIO!=GPM){        // O estado da entrada mudou
53.             if (DataIn==0){     // Se passou para zero,
54.                GPM=0;           // registra em GPM e
55.                Increment();     // incrementa o contador
56.             } else {            // Se passou para 1
57.                GPM=0b1000;      // então apenas registra a mudança
58.             }
59.          }
60.       }
61.    }
62.}
63.
64. //*****************************************************************************
65. // Esta é a rotina que envia os dados para o display. O parâmetro Ind contem
66. // o índice do display que está sendo exibido, variando de 8 a 1. A varredura
67. // é feita da esquerda para a direita. Um zero só é mostrado se for estiver no
68. // display 0 ou se houver um dígito não-zero a sua esquerda. Desta forma evita-
69. // -se mostrar zeros a esquerda, tornando a leitura mais fácil. A cada varredura
70. // são enviados 16 bits aos registradores 74HC595.
71. // Note que Ind pode ser um número entre 1 e 8, mas a matriz do contador varia
72. // de 0 a 3. Para obter o dado correto, Ind deve ser decrementado e dividido por dois.
```

```
73. // Se Ind for impar, toma-se os bits menos significativos e, sendo par, os mais
74. // significativos.
75. //*************************************************************************
76. void SendData(char Ind){
77.   char i = Ind-1;
78.   i >>= 1;                              // Em binário, um shift à direita equivale a
79.                                         // dividir por dois arredondando para baixo
80.   char Sending=Counter[i];
81.   if ((Ind & 0b00000001)==1){           // Se Ind for impar, então imprime os
82.      Sending = Sending & 0b00001111;    // 4 bits menos significativos
83.   } else {
84.      Sending >>= 4;                     // Caso contrário, coloca os 4 bits mais
85.   }                                     // altos na posição de impressão.
86.   if (Sending!=0 || Ind==1) NonZero=1;  // Habilita a impressão de zeros se
87.                                         // encontrar um dígito não-zero ou se
88.                                         // for o primeiro dígito.
89.   if (Sending==0) {Sending = Number0;} else
90.   if (Sending==1) {Sending = Number1;} else
91.   if (Sending==2) {Sending = Number2;} else
92.   if (Sending==3) {Sending = Number3;} else
93.   if (Sending==4) {Sending = Number4;} else
94.   if (Sending==5) {Sending = Number5;} else
95.   if (Sending==6) {Sending = Number6;} else
96.   if (Sending==7) {Sending = Number7;} else
97.   if (Sending==8) {Sending = Number8;} else
98.   {Sending = Number9;}
99.
100.        for(i=8;i!=0;i--){                    // Primeiro envia os bits do display
101.           if ((0b10000000 & Sending)!=0) {   // O bit mais alto vai primeiro
102.              DataOut=1;
103.           }
104.           ClockOut=1;
105.           GPIO=0;
106.           Sending <<= 1;                     // O shift joga o próximo bit na posição
107.        }                                     // para envio
108.        for(i=8;i!=0;i--){
109.           if (i==Ind && NonZero!=0) {DataOut=1;}
110.           ClockOut=1;                        // Quando chegar a posição do dígito a
111.           GPIO=0;                            // ser mostrado, o bit será enviado se
112.        }                                     // algum dígito diferente de zero já
113.                                              // existir no display
114.        ShowData=1;                           // Envia o sinal para copiar os registros
115.        ShowData=0;                           // para a saída
116.     }
117.
118.      //*************************************************************************
119.      // Esta é a rotina que faz a contagem efetivamente. Para simplificar a decodifi-
120.      // cação do display, a contagem é feita em BCD (Binary Coded Decimals). Por este
121.      // padrão, cada grupo de 4 bits do byte conterá um digito de 0 a 9.
122.      // Isso significa que, embora o tipo char possa ir até 255 (FF em hexadecimal),
123.      // a contagem é limitada a 99.
124.      //*************************************************************************
```

```
125.    void Increment(void){
126.      char Carry=1;
127.      char i = 0;
128.      char WorkVal;
129.      while (Carry !=0 && i != 4)       // A soma só termina quando o Carry é
130.      {                                 // zerado ou chegar-se ao fim da matriz.
131.
132.        WorkVal=Counter[i];             // Trabalhar com uma cópia local do valor da
133.                                        // matriz diminui o tamanho final do código
134.                                        // compilado. O PIC10F200 é limitado e as
135.                                        // vezes é preciso gastar RAM para poupar ROM.
136.
137.        if (((WorkVal ^ 9) & 0b00001111)!=0){   // Verifica se os 4 bits menores dão 9.
138.          WorkVal ++;                           // Se não são 9, soma-se 1 e zera
139.          Carry = 0;                            // o carry encerrando o processo.
140.        } else {                                // Caso sejam 9
141.          WorkVal = WorkVal & 0b11110000;       // os bits menores são zerados e
142.          if (WorkVal !=0x90){                  // verifica se os maiores são 9.
143.            WorkVal += 0b00010000;              // Se os 4 bits maiores não são 9,
144.            Carry = 0;                          // soma 1 a eles e zera o carry
145.          }                                     // encerrando o processo.
146.          else{                                 // Se são 9,
147.            WorkVal = 0;                        // zera-os
148.          }                                     // e mantém o carry para continuar o
149.        }                                       // processo.
150.        Counter[i]=WorkVal;                     // Coloca o resultado de volta na
151.        i++;                                    // matriz e passa para o próximo byte.
152.      }
153.    }
```

10 - Código hexadecimal

Abaixo está o código do programa compilado. Crie um arquivo de texto, por exemplo, contador.txt. Copie todo o código apresentado (incluindo dois pontos) neste arquivo texto e o renomeie para contador.hex. O código estará pronto para ser gravado no microcontrolador.

```
:10000000010A75007600770078006400B30A3000BA
:10001000FF0CD00131000304310315OCD101240081
:10002000000232001007180A12020F0E32001B0ADB
:10003000B2030F0C720112024307210AD0004307DA
:10004000230A7900B90212024307280ABE0C580A93
:10005000D20043072D0A880C580A020C920143076C
:10006000330AF40C580A030C92014307390ADC0CDA
:10007000580A040C920143073F0ACA0C580A050C9F
:1000800092014307450A5E0C580A060C9201430789
:100090004B0A7E0C580A070C92014307510A8C0C3C
:1000A000580A080C92014307570AFE0C580ADE0C46
:1000B0003200080C3100110243066A0AF207610A95
:1000C000460506056600030472030 10CB100110227
:1000D00043075E0A080C310011024306800A100231
:1000E0003F0011029F01430779 0A19024306790A6A
:1000F000460506056600010CB100110243076F0AB0
:10010000260526040008700 0B00271001002 4306A4
:10011000000008040C9101430600 08150CD1012400CD
:10012000000232001202090F0F0E4306990A010C59
:10013000A10A1202F00E3200900C92014306A60AA8
:10014000100C3F001F02F2017000A70A7200120299
:100150003F00150CD10124001F022000010C3F00BC
:100160001F02F101860A9F0C0200080C06007300B2
:1001700007900080C340014024306B80A1402FD0981
:10018000013023F0006029F014306CD0A6606CB0A12
:100190007300FE09CD0A080C3300010CB400BB0A41
:0401FA00070A830A63
:021FFE00EBFFF7
:00000001FF
```

11 - Lista de material

Semicondutores

01 - PIC10F200 (PIC1)
02 - 74HC595 (U2, U2)
08 - BC548 (Q1 a Q8)
08 - Displays de 7 segmentos catodo comum (S1 a S8)

Resistores

07 - 330R (R1 a R7)
08 - 10k (R8, R9 e R11 a R16)

Capacitores

02 - 100nF (C1, C2)

Diversos

01 - Chave Táctil (SW1)

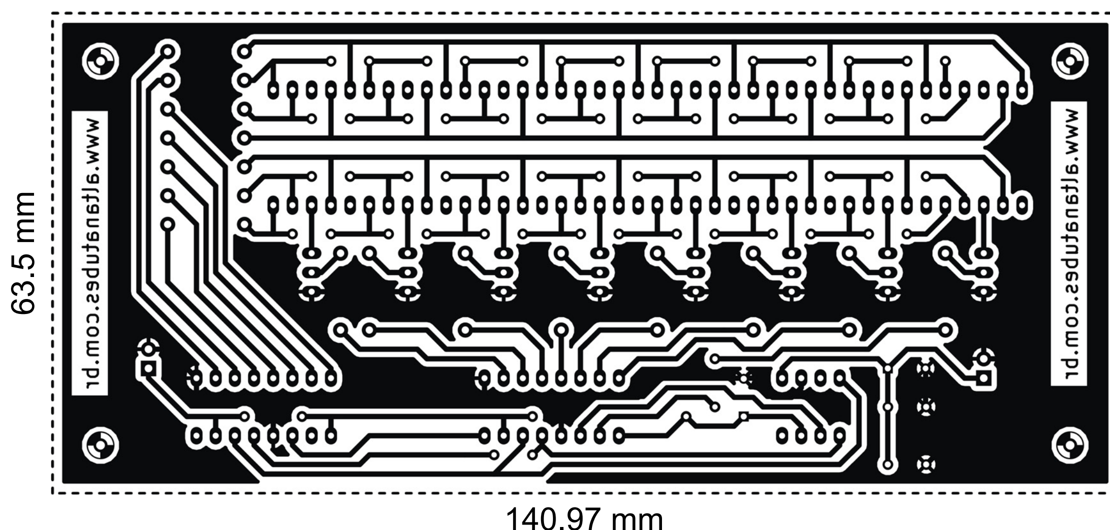
Alimentação entre 3V e 5.5V, ligado em P2
Chave externa para contagem (opcional), ligada em P1

12 - Montagem

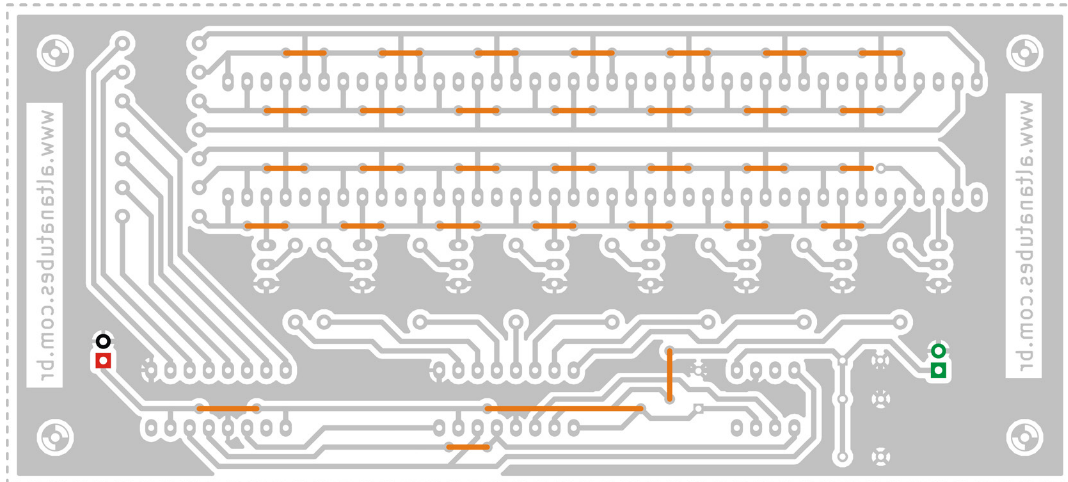
Abaixo é mostrada a imagem da placa, em tamanho natural, bem como a posição dos jumpers e também a posição dos componentes.

Atenção para não esquecer nenhum jumper, especialmente os que vão embaixo dos displays.

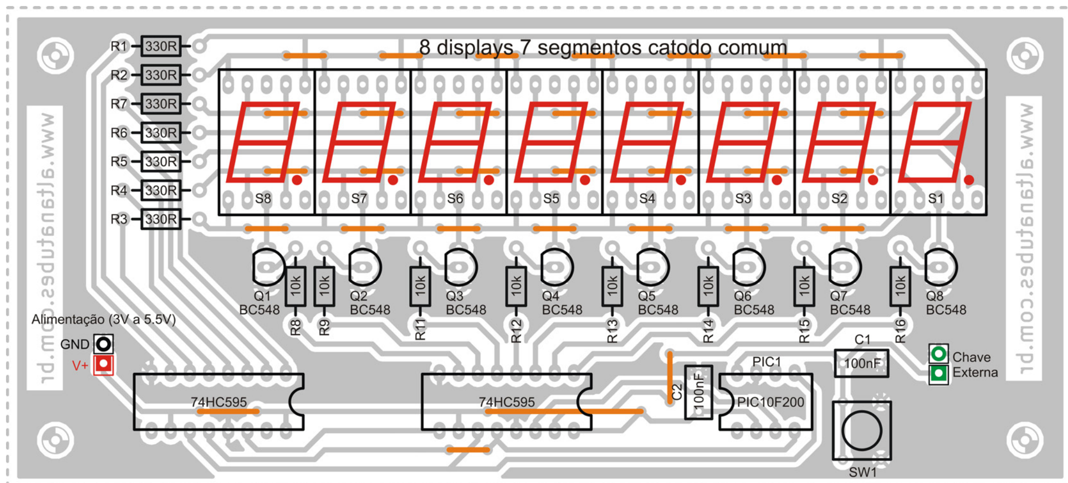
12.1 - Placa



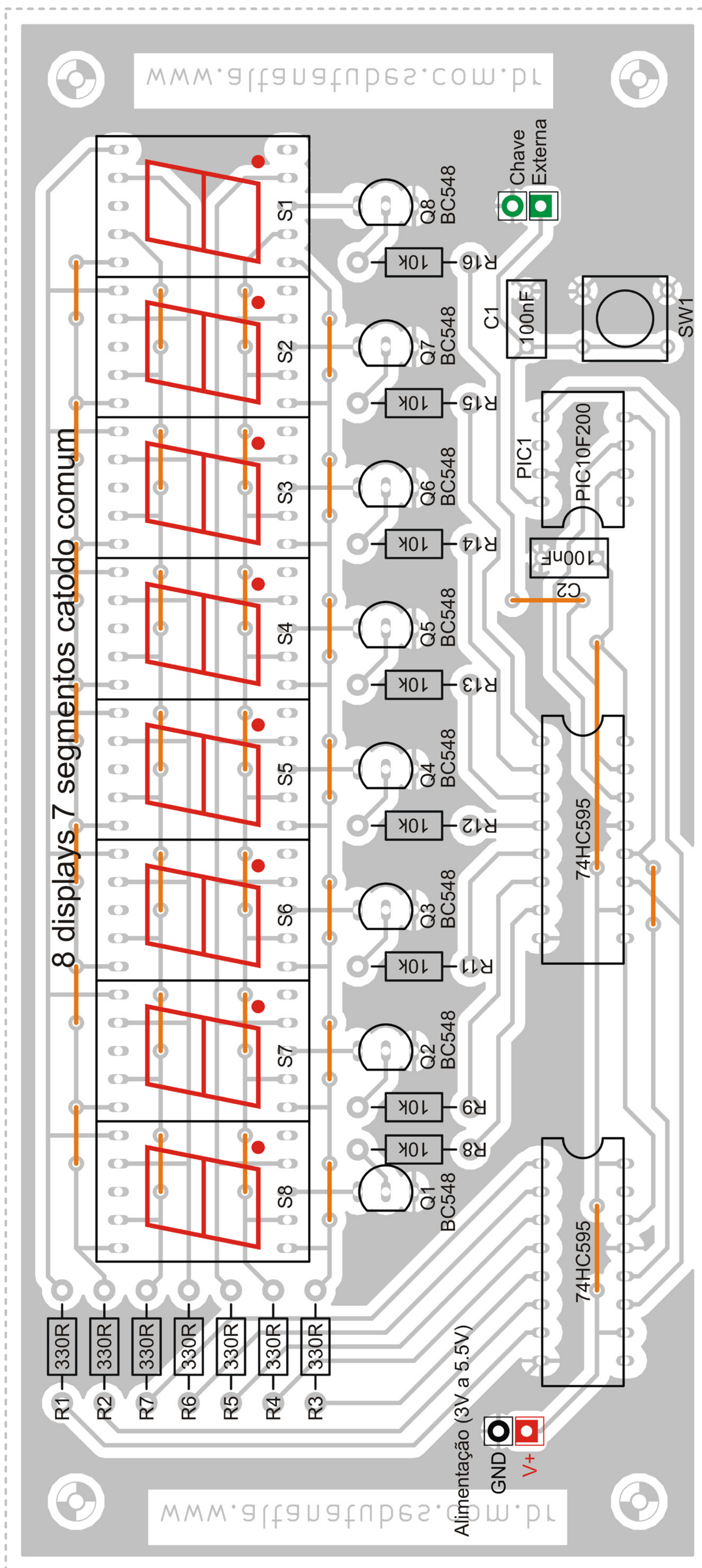
12.2 - Posicionamento dos jumpers



12.3 - Aspecto final da montagem



12.4 - Imagem ampliada da montagem



12.5 - Placas prontas para transferência térmica