

RELATÓRIO:
Programação de percursos para um
carro controlado por motores DC

1/ Introdução e apresentação geral

O Trabalho integrador que apresentamos tem por finalidade expor e ilustrar a lógica e arquitectura utilizadas na programação de percursos a efectuar por implementação de um modelo de carro telecomandado. Para tal implementação usaremos um sistema baseado no microcontrolador 89C51, substituindo a electrónica de controlo e comunicação RF inicialmente presente. O modelo de carro adoptado foi denominado de “PLcar” de modo a simplificar a sua designação técnica.

Facilmente nos apercebemos que a tarefa principal deste projecto seria a programação do microcontrolador e nesse sentido foram desenvolvidos três ficheiros escritos em código assembly que visam diferentes etapas da programação do PLcar. No que diz respeito à programação de percursos propriamente ditos foi usada uma linguagem simples, composta por comandos semelhantes aos da linguagem LOGO, a qual designaremos de PL (Pseudo-LOGO).

Para que seja possível criar percursos em código fonte PL, é necessário atribuir uma relação entre cada comando ao seu respectivo código objecto em hexadecimal. Nesse sentido foi criada uma tabela que comanda um assembler universal, o TASM, que por sua vez converte o código fonte em PL para código objecto. O envio deste código para o microcontrolador instalado no PLcar é assegurado pela transmissão via RS-232C, que em conjunto com o seu conversor de nível, constitui a mais generalizada forma de comunicação série.

O PLcar está munido de dois leds de sinalização, sendo estes, o “led_ok”, que sinaliza a recepção de RS-232C com sucesso e o “lederr” que por sua vez sinaliza a detecção de um erro. O botão “PLAY”, também presente no circuito de controlo do PLcar, ao ser premido, dará início à execução do código objecto, melhor dizendo, do percurso, assim que o “led_ok” estiver aceso. Toda esta lógica de funcionamento, assim como a análise pormenorizada de sequências, sai com certeza da temática introdutória, dando lugar a um capítulo específico deste relatório, o capítulo quatro, onde é estudada a lógica e descrição do software.

2/ Descrição do hardware

O circuito de controlo do PLcar é constituído por vários componentes, entre os quais se destacam três componentes principais que desempenham as seguintes funções:

i. Microcontrolador 89C51:

O microcontrolador, disponível com memória de programa do tipo flash, é responsável por toda a programação do PLcar assim como pelo controlo de todas as componentes do circuito;

ii. Conversor de nível MAX 232:

Converte os níveis lógicos do microcontrolador para os +/- 12V da ligação RS-232C;

iii. ST L293E:

Circuito que controla os dois motores do PLcar.

Todos os componentes do circuito de controlo e interacção entre os mesmos estão representados no diagrama de blocos (diagrama 1). No que diz respeito ao controlo de movimento dos motores, este deve ser efectuado de acordo com as indicações apresentadas na tabela 1:

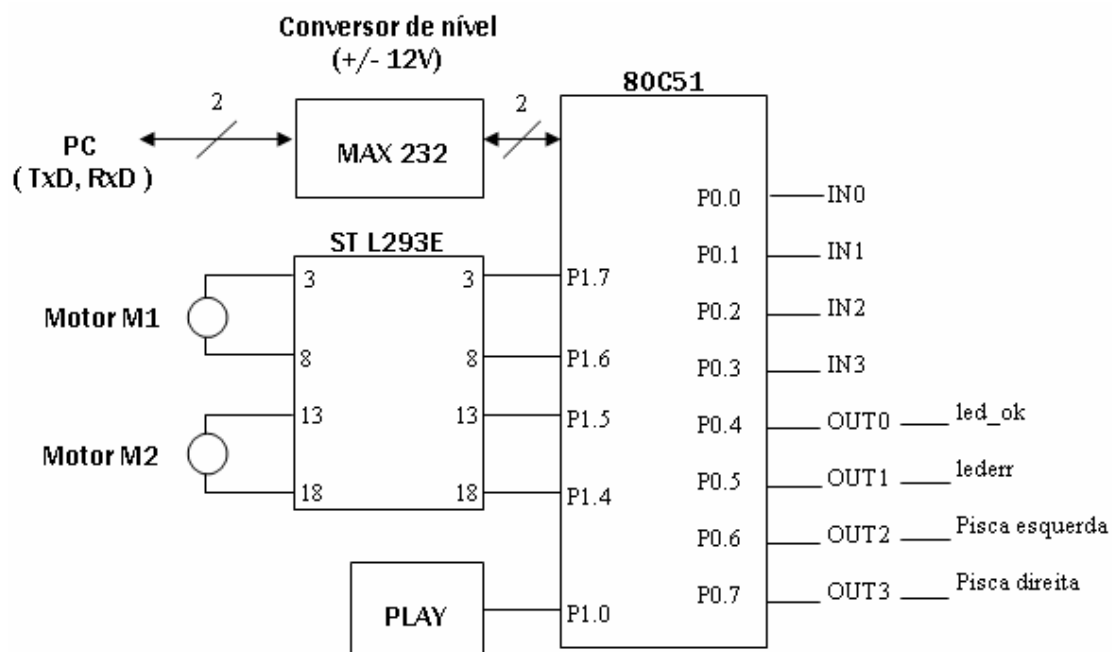


Diagrama 1: Diagrama de blocos do circuito de controlo do PLCar

Motor de direcção (M1)	Motor de accionamento (M2)
Esquerda: Pino 2: H; Pino 9: L	Frente: Pino 12: H; Pino 19:L
Direita: Pino 2: L; Pino 9: H	Trás: Pino 12: L; Pino 19: H
Parado: Pino 2 = Pino 9	Parado: Pino 12 = Pino 19

Tabela 1: Controlo de movimento do PLCar

3/ Programação do percurso

Como já foi referido na introdução, a programação de percursos para o PLcar assenta numa dicotomia entre a criação do código fonte PL e a sua respectiva representação em código objecto.

Para tornar viável essa relação, foi criada uma tabela, a qual denominamos de “Tabela de Opcodes para o TASM”, que pode ser visualizada no final deste relatório, em anexo, no capítulo 5/. A tabela de Opcodes desempenha assim um papel fundamental na programação de percursos, já que é em função da mesma que o assembler universal TASM é controlado.

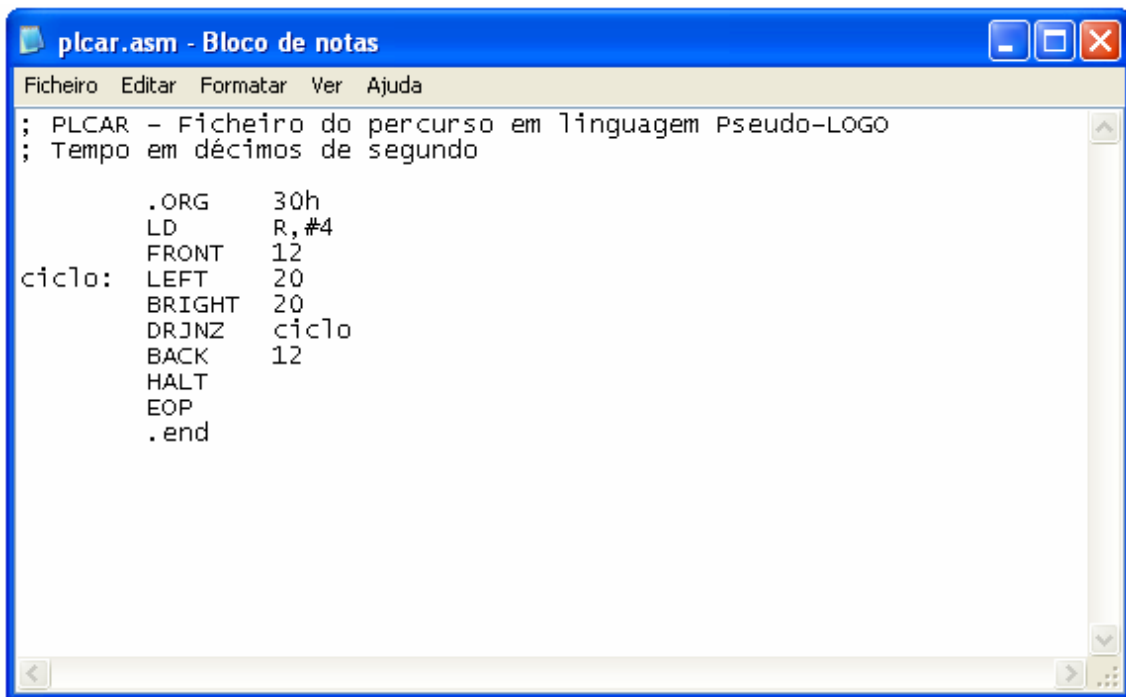
Se analisarmos a tabela, é facilmente perceptível a sua divisão em colunas, onde é possível distinguir: o *nome de cada instrução* (INSTR), os *argumentos utilizados em cada uma das instruções* (ARGS), as *suas respectivas representações hexadecimais em código ASCII* (OPCODES), o *número de bytes utilizados por cada instrução* (BYTES), o *módulo* (MODULE), a *classe* (CLASS), o *nome dado a cada segmento* (SEGMENTO) e por último, a *representação alfabética de cada instrução em código ASCII* (ASCII).

A escrita do código fonte PL será feita com base nos comandos apresentados na seguinte tabela:

Comando PL	Descrição
FRONT Δt	Anda para trás durante Δt décimos de seg. ([0..127])
BACK Δt	Anda para trás durante Δt décimos de seg. ([0..127])
RIGHT Δt	Vira para a direita durante Δt décimos de seg. ([0..127])
LEFT Δt	Vira para a esquerda durante Δt décimos de seg. ([0..127])
BRIGHT Δt	Anda para trás e vira para a direita durante Δt décimos de seg. ([0..127])
BLEFT Δt	Anda para trás e vira para a esquerda durante Δt décimos de seg. ([0..127])
SETOUTn	Coloca a saída n em 1 (n = 0..3)
RSTOUTn	Coloca a saída n em 0 (n = 0..3)
LD R,VAL	Carrega o registo R com VAL ([0..255])
DRJNZ ADDR	Decrementa R e salta para ADDR ([0..255]) se R \neq 0
JPNINk ADDR	Salta para o endereço ADDR ([0..255]) se a entrada k estiver em 0 (k=0..3)
JP ADDR	Salta para o endereço ADDR ([0..255])
HALT	Pára o movimento
EOP (End Of Program)	Termina o código fonte (não é verdadeiramente um comando: a recepção dos códigos ASCII 'E', 'O' e 'P', assinala o fim do ficheiro fonte PL)

Tabela 2: Descrição dos comandos PL suportados.

A título de melhor ilustrar a facilidade da escrita de percursos em código PL, de seguida expomos um exemplo de um percurso facilmente realizável pelo PLcar:

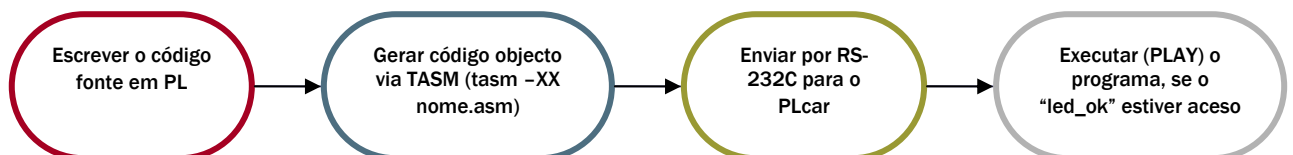


```
plcar.asm - Bloco de notas
Ficheiro Editar Formatar Ver Ajuda
; PLCAR - Ficheiro do percurso em linguagem Pseudo-LOGO
; Tempo em décimos de segundo

        .ORG      30h
        LD        R, #4
        FRONT    12
ciclo:  LEFT     20
        BRIGHT   20
        DRJNZ    ciclo
        BACK     12
        HALT
        EOP
        .end
```

Exemplo: Percurso escrito em código PL

De forma a concluir este capítulo, foi criado um fluxograma no sentido de rever, passo a passo, as etapas pelas quais deverá passar a criação de um percurso:



Fluxograma 1: Etapas no desenvolvimento de um percurso

4/ Lógica e descrição do software

Com o intuito de explicitar todo o estudo desenvolvido no âmbito da programação do PLcar, foi efectuada uma partição deste capítulo em três sub temas que visam naturalmente diferentes objectos de análise e estudo (i. Funcionamento de uma interrupção; ii. Funcionamento dos “Leds”; iii. Lógica do Timer0). Com a ajuda dos fluxogramas apresentados assim como pela divisão temática efectuada, todo o software desenvolvido poderá ser compreendido na sua totalidade.

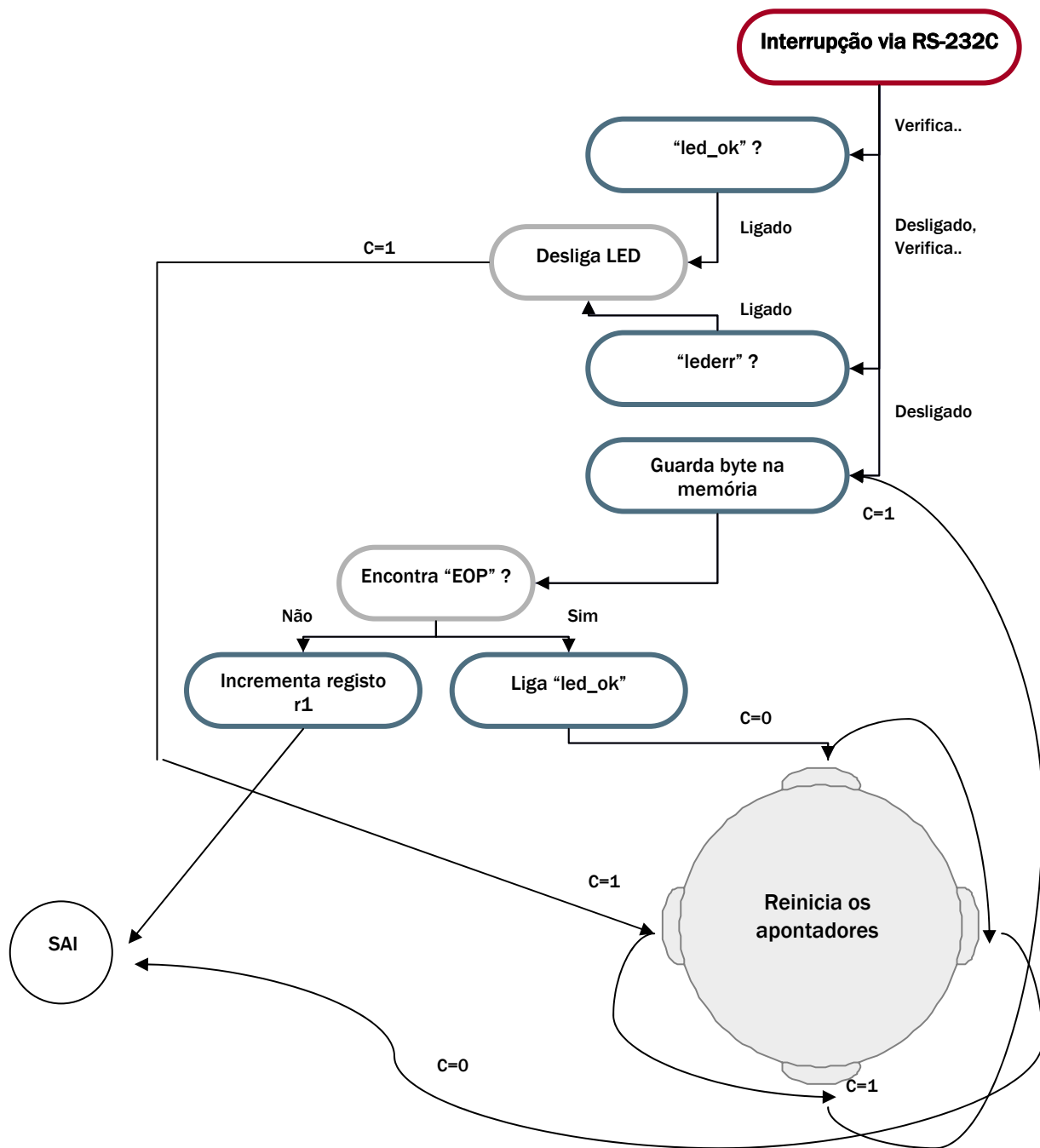
Contudo, é importante referir à partida, como nota introdutória, que a programação do microcontrolador assenta basicamente na criação de três ficheiros, sendo: o primeiro, que trata do código principal e do interpretador de comandos suportados, o qual denominamos de “INTRPRET.A51”; o “COMMANDS.A51”, ficheiro com os comandos a serem executados pelo interpretador, que contém essencialmente as instruções que definem cada um dos comandos; e por último, o ficheiro que trata do código das interrupções da porta série e do timer0, o qual denominamos de “INTERRUP.A51”.

Todos estes ficheiros de código poderão ser consultados em anexo, no capítulo 5/.

De forma a evitar uma descrição extensa e pormenorizada do conteúdo de cada ficheiro, foram adicionados comentários no próprio código, que analisados em conjunto com este, possibilitam a melhor forma de compreensão do programa, até mesmo ao nível de sucedimento lógico e ordem de desenvolvimento.

i. Funcionamento de uma interrupção

O ficheiro “*interrup.a51*” comanda o funcionamento das 2 interrupções existentes, que são: a *interrupção da porta série*, que controla a entrada de novos percursos; e a *interrupção do timer0*, que define o tempo de movimento do carro. Para explicar o comportamento da primeira interrupção, respectivamente enunciada, desenhamos um fluxograma (Fluxograma 2). No que diz respeito à segunda interrupção, está explicitado todo o seu funcionamento no terceiro subtema deste capítulo, que trata exclusivamente da lógica do Timer0.



Fluxograma 2: Interrupção via comunicação série RS-232C

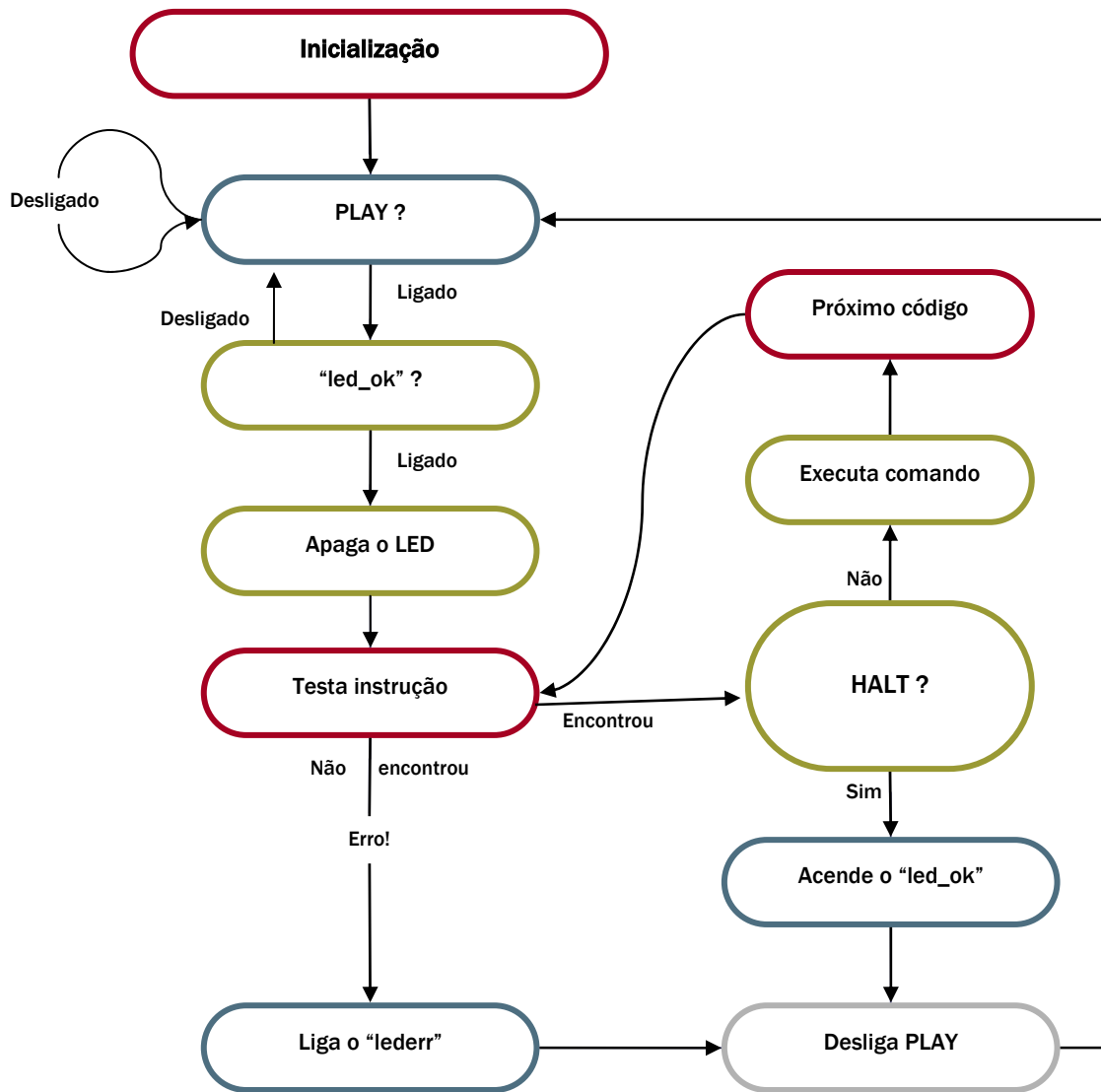
ii. Funcionamento dos “Leds”

Numa fase inicial, pareceu-nos como melhor opção, colocar os dois “leds” apagados até ao momento de recepção de um programa. Quando o PLCar recebe um programa, o “led_ok” acende-se apenas quando for lido o segmento “EOP” (End Of Program) no código objecto enviado. Este fica aceso até se carregar no PLAY. Quando o PLCAR chegar ao fim do percurso para o qual foi programado, o “led_ok” acender-se-á de novo.

Caso surja alguma irregularidade (p.e., algum comando que não seja reconhecido), o “lederr” acende-se, apaga-se o “led_ok” e desliga-se o PLAY. O “lederr” só se apaga aquando da recepção de um novo percurso. No caso de o PLCar terminar o seu percurso sem quaisquer irregularidades, desligar-se-á o play e acender-se-á o “led_ok”, permanecendo ligado até que um novo programa seja recebido, ou que o PLAY seja carregado de novo.

Quando for detectada uma nova recepção pela porta série, o “led_ok” vai apagar-se (tal como já foi dito em cima) até ao momento em que encontrar um “EOP” no novo programa.

Com a ajuda deste fluxograma, é possível relacionar o funcionamento dos leds com o interpretador de comandos e também entender todo o seu modo de funcionamento (Fluxograma 3):



Fluxograma 3: Interpretador de comandos e funcionamento dos leds

iii. Lógica do Timer0

Os comandos de movimento do PLCar recebem argumentos em décimos de segundo, que será a duração do seu deslocamento. Esse tempo é gerado pelo timer0. Se configurarmos o timer para um modo de 16 bits, o máximo de tempo que conseguimos gerar são 0,065535s. Desta forma, usa-se 0,05s como base de tempo pois $2 \times 0,05 = 0,1$. Isto diz-nos que para gerar um décimo de segundo serão precisos dois overflows do timer. Nesta lógica basta multiplicar o número de décimos de segundo fornecidos pelo argumento do código do percurso por dois e teremos o número de overflows necessários para atingir o tempo pedido.

Abaixo representamos os cálculos efectuados para chegar a estas conclusões:

$$\frac{\text{freq. osc.}}{\text{ciclos de relógio}} = \frac{12 \times 10^6}{12} = 10^6 \text{ incrementos/s}$$

Timer 16: incrementa até 65535

No máximo... $\frac{65535}{10^6} = 0,065535$ segundos até ao overflow

Definir para 0,05 s

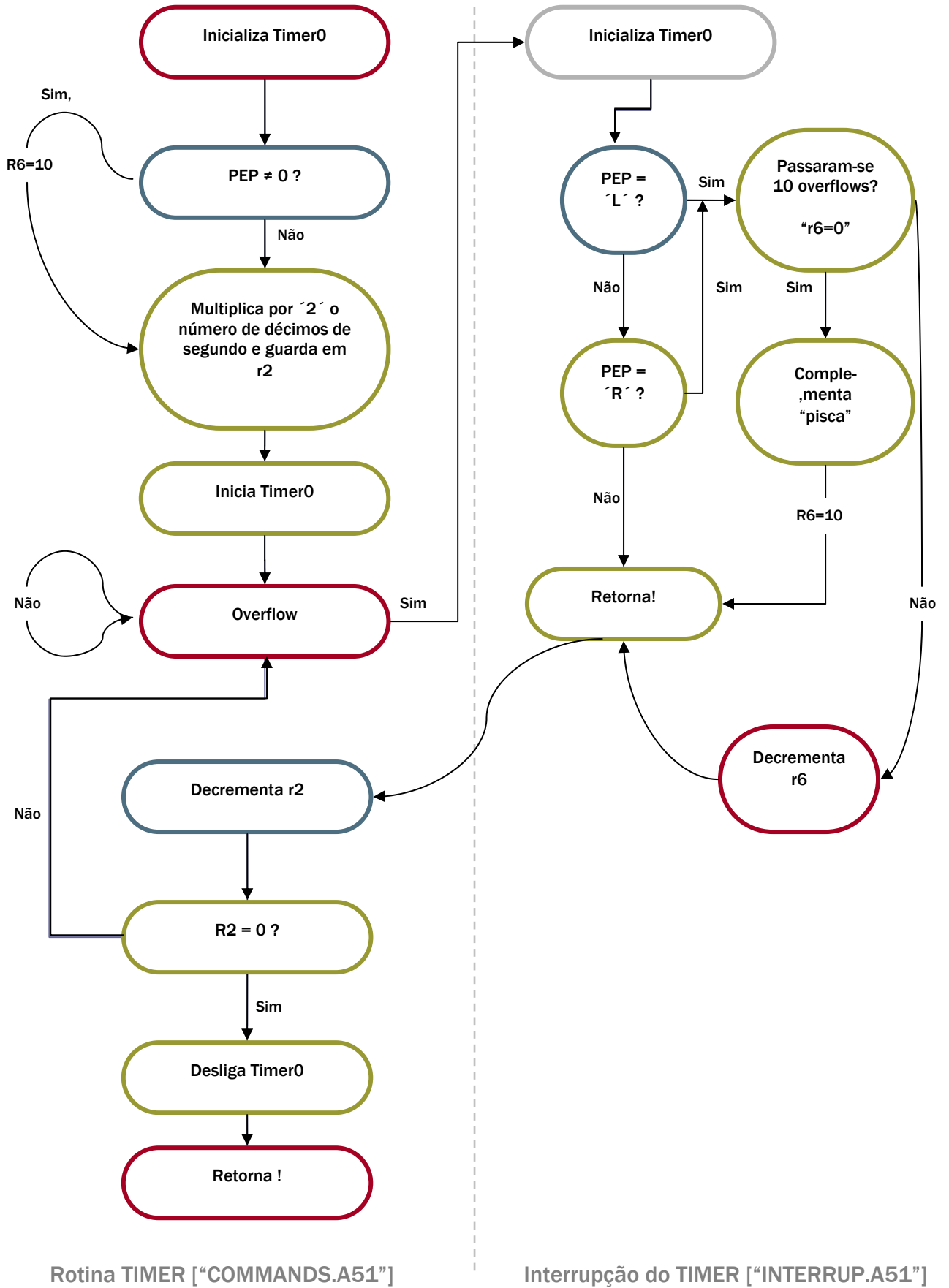
Quantas vezes é incrementado?

$0,05 \times 10^6 = 50000$ incrementos

TH0 = #high(65535 – 50000)

TL0 = #low(65535-50000)

Serão então usados esses valores do TH0 e TL0 para configurar o Timer0, gerando um intervalo de tempo de 0,05 seg. Na próxima página apresentamos um fluxograma de forma a explicar como funciona a rotina timer, responsável por gerar o tempo de base para os movimentos do PLCar.



Fluxograma 4: Funcionamento da rotina TIMER

5/ Anexos

Reservámos este capítulo para a inclusão da tabela de Opcodes para o TASM assim como dos três ficheiros assembly, respectivamente “INTRPRET.A51”, “COMMANDS.A51” e “INTERRUPT.A51”.

```
"PLCAR - Tabela de opcodes para o TASM"
/* O uso de códigos ASCII nos opcodes facilita a simulação no dScope */

/* INSTR      ARGS  OPCODE BYTES  MODULE CLASS  SEGMENTO      ASCII  */
FRONT         *      46      2      NOP      1      /* front      F      */
BACK          *      42      2      NOP      1      /* back       B      */
RIGHT         *      52      2      NOP      1      /* right      R      */
BRIGHT        *      29      2      NOP      1      /* bright     )      */
LEFT          *      4C      2      NOP      1      /* left       L      */
BLEFT         *      28      2      NOP      1      /* bleft      (      */
SETOUT0       ""      30      1      NOP      1      /* sto0       0      */
SETOUT1       ""      31      1      NOP      1      /* sto1       1      */
SETOUT2       ""      32      1      NOP      1      /* sto2       2      */
SETOUT3       ""      33      1      NOP      1      /* sto3       3      */
RSTOUT0       ""      36      1      NOP      1      /* rsto0      6      */
RSTOUT1       ""      37      1      NOP      1      /* rsto1      7      */
RSTOUT2       ""      38      1      NOP      1      /* rsto2      8      */
RSTOUT3       ""      39      1      NOP      1      /* rsto3      9      */
LD            R,#*    6C      2      NOP      1      /* load       l      */
DRJNZ         *      44      2      NOP      1      /* drjnz      D      */
JPNIN0        *      61      2      NOP      1      /* jpn0       a      */
JPNIN1        *      62      2      NOP      1      /* jpn1       b      */
JPNIN2        *      63      2      NOP      1      /* jpn2       c      */
JPNIN3        *      64      2      NOP      1      /* jpn3       d      */
JP            *      4A      2      NOP      1      /* jp         J      */
HALT          ""      48      1      NOP      1      /* halt       H      */
EOP           ""      454F50 3      NOP      1      /* não é um comando*/
```

Anexo 1: Tabela de Opcodes para o TASM

```

; Código principal + interpretador de comandos
; -----

play equ    p1.0
led_ok equ  p0.4
lederr equ  p0.5

vars segment data
    rseg vars
addr: ds    1
pep: ds    1

public addr, pep
extrn code(front, back, right, bckr, left, bckl)
extrn code(sto0, stol, sto2, sto3, rsto0, rsto1, rsto2, rsto3)
extrn code(load, drjnz, jpn0, jpn1, jpn2, jpn3, jp, halt)

    cseg at 0
    jmp interp

interp segment code ; código principal
    rseg interp ; começa por inicializar os registos
    mov sp, #0Fh ; para proteger as vars da stack
    mov ie, #93h ; permite ints. ES, ET0 e EX0
    mov scon, #50h ; porta série em modo 1, recepção habilitada
    orl pcon, #80h ; SMOD=1
    mov tmod, #21h ; Timer 1 em modo 2 - Timer 0 em modo 1
    mov th1, #0FDh ; Baud Rate = 19.200 bps
    mov tl1, #0FDh
    setb tr1 ; activa T/C 1
    mov pep, #00h
    mov r0, #30h ; apontador para execução do interpretador
    mov r1, #30h ; apontador para recepção de bytes via RS232
                    ; aqui começa o interpretador de comandos
init: setb ren ; aceita novos bytes para novo percurso
    jb play, $ ; início do interpretador de comandos
    jnb led_ok, cont ; se o led OK nao estiver aceso, para o prog
    jmp stop
cont: setb led_ok ; mal inicia a execução mantém os 2 leds desl.
    clr ren ; e impede recepção de novos bytes
test: mov a, @r0 ; começa a fase de teste do comando a executar
    jnb play, tfront ; para poder parar o carro num ciclo infinito
    jmp shalt
tfront:
    cjne a, #'F', tback ; se não for este comando, testa o próximo
    acall front
    jmp next
tback: cjne a, #'B', tright
    acall back
    jmp next
tright:
    cjne a, #'R', tbr
    acall right
    jmp next
tbr: cjne a, #')', tleft
    acall bckr
    jmp next
tleft: cjne a, #'L', tbl
    acall left
    jmp next
tbl: cjne a, #'(', tsto0
    acall bckl
    jmp next
tsto0: cjne a, #'0', tsto1
    acall sto0
    jmp next
tsto1: cjne a, #'1', tsto2
    acall stol
    jmp next
tsto2: cjne a, #'2', tsto3

```

```

        acall  sto2
        jmp   next
tsto3:  cjne  a,#'3',trsto0
        acall  sto3
        jmp   next
trsto0:
        cjne  a,#'6',trsto1
        acall  rsto0
        jmp   next
trsto1:
        cjne  a,#'7',trsto2
        acall  rsto1
        jmp   next
trsto2:
        cjne  a,#'8',trsto3
        acall  rsto2
        jmp   next
trsto3:
        cjne  a,#'9',tload
        acall  rsto3
        jmp   next
tload:  cjne  a,#'l',tdrjnz
        acall  load
        jmp   next
tdrjnz:
        cjne  a,#'D',tjpn0
        acall  drjnz
        jmp   next
tjpn0:  cjne  a,#'a',tjpn1
        acall  jpn0
        jmp   next
tjpn1:  cjne  a,#'b',tjpn2
        acall  jpn1
        jmp   next
tjpn2:  cjne  a,#'c',tjpn3
        acall  jpn2
        jmp   next
tjpn3:  cjne  a,#'d',tjp
        acall  jpn3
        jmp   next
tjp:    cjne  a,#'J',thalt
        acall  jp
        jmp   next
thalt:  cjne  a,#'H',error      ; se não há comando, código não existe (erro)
shalt:  acall  halt            ; se encontra halt, o programa chegou ao fim
stop:   setb  play            ; pára o programa ao chegar a init
        jmp   init
error:  clr   lederr          ; se há erro, liga o led e desliga play ...
        sjmp  stop            ; ... para esperar novo percurso válido
next:   inc   r0              ; passa ao próximo código a testar
        jmp   test
end

```

Anexo 2: Ficheiro assembly “INTRPRET.A51”

```

; Ficheiro com os comandos a serem executados pelo interpretador.
; 1) Tempo usado por consenso em décimos de segundo (máximo 12,8 segs).
; 2) O registo R está definido no registo R3.
; 3) R6 será usado para contar o tempo para os piscas
; 4) pep (palavra de estado para piscas): 4Ch-(L)eft, 52h-(R)ight, 00h-n/a
;   é suposto os piscas ficarem intermitentes de meio em meio segundo

```

```

IN0    equ    p0.0
IN1    equ    p0.1
IN2    equ    p0.2
IN3    equ    p0.3
OUT0   equ    p0.4
OUT1   equ    p0.5
OUT2   equ    p0.6
OUT3   equ    p0.7
pino2  equ    p1.7
pino9  equ    p1.6
pino12 equ    p1.5
pino19 equ    p1.4

```

```

public front,back,right,bckr,left,bckl
public sto0,sto1,sto2,sto3,rsto0,rsto1,rsto2,rsto3
public load,drjnz,jpn0,jpn1,jpn2,jpn3,jp,halt
extrn data(addr,pep)

```

```

cmds    segment      code          ; esta rotina é a responsável por gerar o
rseg    rseg         cmds         ; o tempo utilizado para mover o plcar

timer:  push    acc
        mov     TH0,#high(65535-50000)
        mov     TL0,#low(65535-50000)
        mov     a,pep              ; identifica se é necessário inicializar r6
        jz      np
        mov     r6,#10             ; 10x0.05=0.5 segs
np:     mov     a,@r0              ; timer atinge overflow em 0,05 s, logo
        mov     b,#02h            ; atinge 0,1 s em cada 2 overflows
        mul     ab                 ; nesta lógica, multiplica-se por 2 o número
        mov     r2,a              ; de déc de seg dado e obtem-se o pretendido
        setb    TR0               ; inicia o timer 0
delay:  jnb     TF0,$              ; espera pelo overflow do timer
        djnz   r2,delay           ; executa 2n vezes
        clr     TR0
        pop     acc
        ret

front:  inc     r0                 ; segundo byte do comando
        setb    pino12            ; faz andar
        clr     pino19            ; para a frente
        acall   timer             ; gera o intervalo de tempo pretendido
        setb    pino19            ; pino12=pino19 >> pára
        jmp     exit

back:   inc     r0
        clr     pino12
        setb    pino19
        acall   timer
        setb    pino12
        jmp     exit

left:   inc     r0
        mov     pep,#'L'
        setb    pino12            ; para a frente
        clr     pino19
        setb    pino2             ; para a esquerda
        clr     pino9
        acall   timer
        setb    OUT2              ; apaga pisca caso esteja aceso
        setb    pino9             ; pára
        setb    pino19
        jmp     exit

```

```

bckl: inc    r0
      mov    pep, #'L'
      clr    pino12           ; para trás
      setb   pino19
      setb   pino2           ; para a esquerda
      clr    pino9

      acall  timer
      setb   OUT2
      setb   pino12         ; pára
      setb   pino9
      jmp    exit

right: inc    r0
      mov    pep, #'R'
      setb   pino12         ; para a frente
      clr    pino19
      clr    pino2           ; para a direita
      setb   pino9
      acall  timer
      setb   OUT3
      setb   pino19         ; pára
      setb   pino2
      jmp    exit

bckr: inc    r0
      mov    pep, #'R'
      clr    pino12         ; para trás
      setb   pino19
      clr    pino2           ; para a direita
      setb   pino9
      acall  timer
      setb   OUT3
      setb   pino12         ; pára
      setb   pino2

exit:  mov    pep, #0        ; apaga as palavras de estado
      ret

sto0:  setb   OUT0
      ret
sto1:  setb   OUT1
      ret
sto2:  setb   OUT2
      ret
sto3:  setb   OUT3
      ret
rsto0: clr    OUT0
      ret
rsto1: clr    OUT1
      ret
rsto2: clr    OUT2
      ret
rsto3: clr    OUT3
      ret

load:  inc    r0
      mov    03h, @r0        ; move novo valor para R3
      ret

drjnz: inc    r0
      dec    r3              ; R3=R
      push  acc
      mov   a, r3
      jz    izero           ; se R=0 sai
      mov   addr, @r0
      mov   r0, addr
      dec   r0              ; para compensar o inc r0 na saída
izero: pop    acc
      ret

```

```

jpn0: inc    r0                ; jpnk - salta para k se INk não for 0
      jb    IN0,sai0
      mov   addr,@r0
      mov   r0,addr
      dec   r0
sai0:  ret
jpn1:  inc    r0
      jb    IN1,sai1
      mov   addr,@r0
      mov   r0,addr
      dec   r0
sai1:  ret
jpn2:  inc    r0
      jb    IN2,sai2
      mov   addr,@r0
      mov   r0,addr
      dec   r0
sai2:  ret
jpn3:  inc    r0
      jb    IN3,sai3
      mov   addr,@r0
      mov   r0,addr
      dec   r0
sai3:  ret

jp:    inc    r0                ; salta para um endereço
      mov   addr,@r0
      mov   r0,addr
      dec   r0                ; para compensar inc r0 à saída
      ret

halt:  mov    r0,#30h           ; reinicializa apontador e
      clr   OUT0              ; OUT0=led_ok: permite voltar a executar
      ret

end

```

Anexo 3: Ficheiro assembly "COMMANDS.A51"

```

; Ficheiro com o código das interrupções da porta série e timer 0
; -----

led_ok equ    p0.4
lederr equ    p0.5
piscae equ    p0.6
piscad equ    p0.7
pino2  equ    p1.7
pino9  equ    p1.6
pino12 equ    p1.5
pino19 equ    p1.4
obstac equ    p3.2

extrn data(peg)
extrn code(bckr)

int_ob      segment    code
receiv      segment    code
timerf0     segment    code

; atendimento das chamadas de cada interrupção

        cseg    at 000Bh
        jmp     timerf0

        cseg    at 0023h
        jmp     receiv

; O carry é usado para indicar se o byte recebido é guardado ou não
; Não é necessário guardar psw pois não haverá conflito com o uso de C

        rseg    receiv          ; interrupção porta série
        clr     ri
        clr     c
        cjne   r0,#7Fh,ok       ; verifica o espaço ocupado
        setb   led_ok
        clr    lederr
        sjmp   reset
ok:     jb     led_ok,err       ; verifica led_ok (se desligado ver lederr)
        setb   led_ok          ; com led_ok, permite receber novo percurso
        setb   c               ; para indicar que deve guardar o byte
        sjmp   reset          ; para isso reinicializa-se o apontador
err:    jb     lederr,save      ; verifica lederr
        setb   lederr          ; se a interrupção surgiu em estado de erro,
        setb   c               ; reinicializa, apaga o led e recebe novo byte
        sjmp   reset
save:   mov    @r1,sbuf
        cjne   @r1,'P',seq1
        dec    r1
        cjne   @r1,'#0',seq2
        dec    r1
        cjne   @r1,'#E',seq3
        clr    led_ok          ; e acende led_ok..pronto p carregar no play
reset:  mov    r1,#30h          ; chegou ao fim,entao reinicializa apontadores
        mov    r0,#30h
        jc     save            ; guarda valor se c=1
        sjmp   sai
seq3:   inc    r1
seq2:   inc    r1
seq1:   inc    r1
sai:    reti

; O registo R6 é usado para gerar os piscas
; Os piscas são activados interminantemente durante a viragem

        rseg    timerf0          ; interrupção de overflow do timer 0
        mov    TH0,#high(65535-50000) ; para gerar 0,05 s
        mov    TL0,#low(65535-50000)

        push   acc
        mov    a,peg

```

```

        cjne  a,#'L',chkpd      ; verifica se é necessário ligar o
        djnz  r6,gx            ; pisca esquerdo (guardado em pep)
        cpl   piscae
        mov   r6,#10
chkpd:  cjne  a,#'R',gx        ; mesmo para o pisca direito
        djnz  r6,gx
        cpl   piscad
        mov   r6,#10
gx:     pop   acc
        reti

end

```

Anexo 4: Ficheiro assembly “INTERRUP.A51”