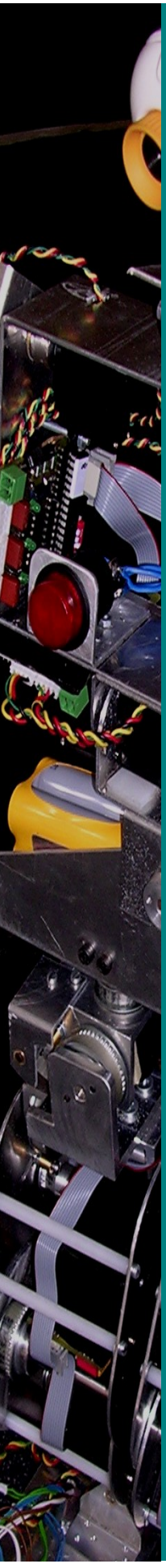


Desenvolvimento de Algoritmos de Controlo para Locomoção de um Robot Humanóide

Manual do Programador

Milton Ruas da Silva Nº21824

Orientação:

Prof. Dr. Filipe Silva (DETI-IEETA)
Prof. Dr. Vítor Santos (DEM-TEMA)

Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática, IEETA
Licenciatura em Engenharia Electrónica e Telecomunicações

Agosto de 2006

Índice

1. Apresentação

Este manual destina-se ao programador que pretenda trabalhar no software de baixo nível das placas de controlo Master e Slaves, bem como também nos scripts MatLab desenvolvidos para comunicação e visualização de dados.

Toda a documentação descrita neste manual reporta-se à versão 2.00 desenvolvida no ano lectivo 2005/06.

2. Programa MASTER

Localização: **<CD_PROJ>:\Lab\Fase3_Integration\Last Version\Master_Stable_2.00**

As relações de inclusão entre os diversos módulos estão esquematizadas na Fig. 1. Cada módulo representa um par de ficheiros .h e .c com o protótipo das funções externas e a sua implementação respectivamente. Apenas o módulo Master apenas possui um ficheiro .c dado que é o que contém a função *main*.

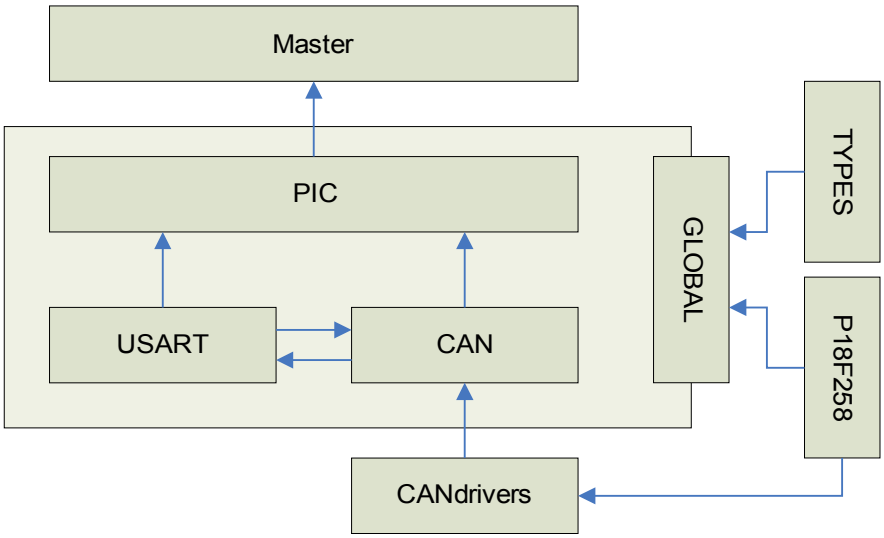


Fig. 1: Relações de inclusão dos módulos de software do Master.

Módulo Master

Este módulo é o primeiro a ser executado (função *main*) e é responsável por fazer a chamada das funções de configuração presentes no módulo PIC. Após os procedimentos de configuração, fica preso numa *dummy task* em que o único software em execução é proveniente da rotina de serviço à interrupção definida no módulo PIC.

Módulo PIC

Módulo com a implementação da rotina de inicialização do PIC e da rotina de serviço às interrupções provenientes da USART e do CAN. É neste módulo que são inicializados e implementados os mecanismos de comunicação com a unidade principal e as unidades slave.

Tabela 1: Funções do módulo PIC.

Funções	***Descrição***
initPic	Inicialização dos periféricos associados às comunicações USART e CAN.
isr	Implementação dos mecanismos de comunicação pela USART com a unidade principal e da comunicação CAN com as unidades slave.

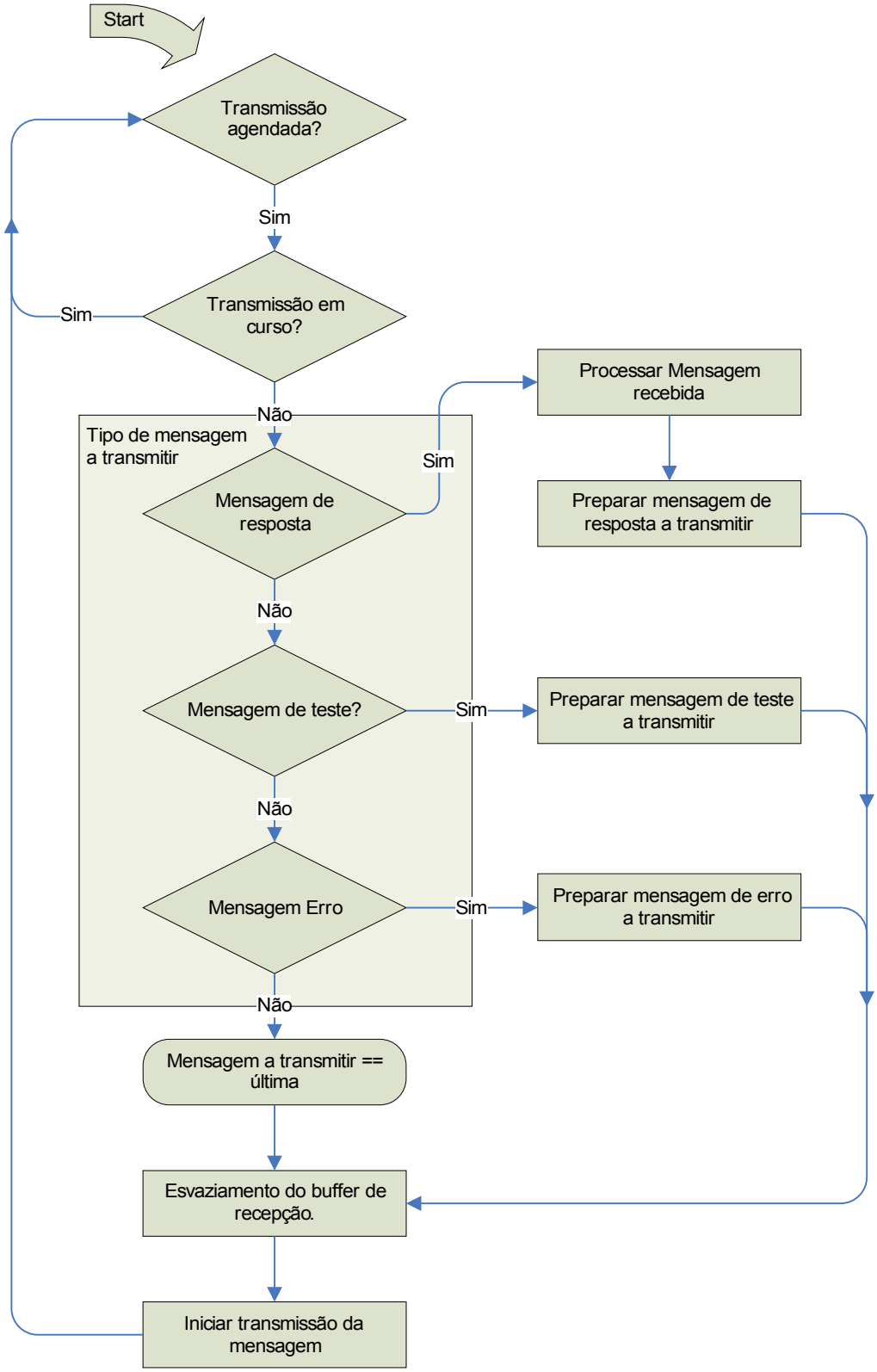


Fig. 2: Algoritmo de recepção de informação, via USART, pelo Master

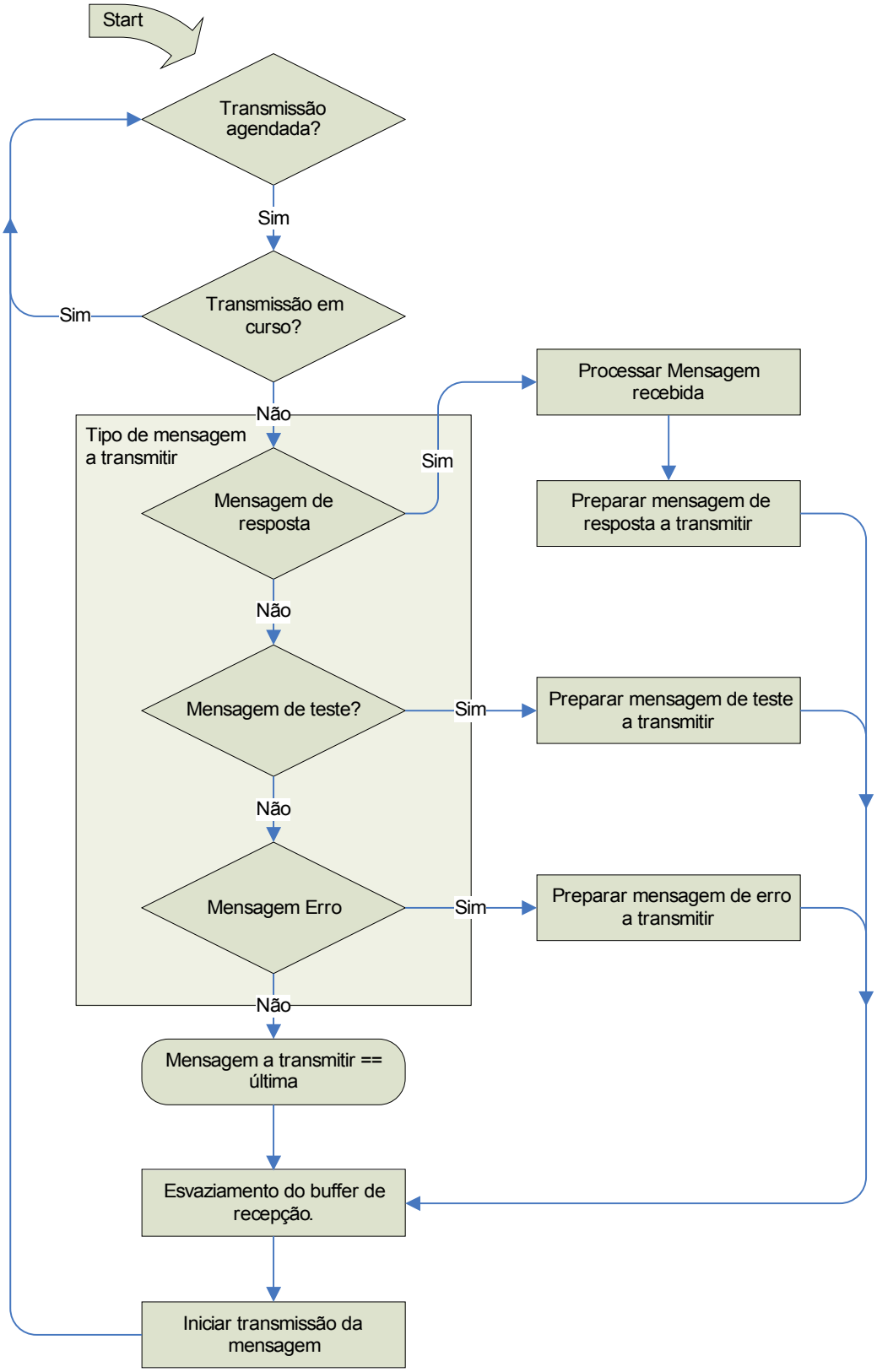


Fig. 3: Algoritmo de transmissão de informação, via USART, pelo Master.

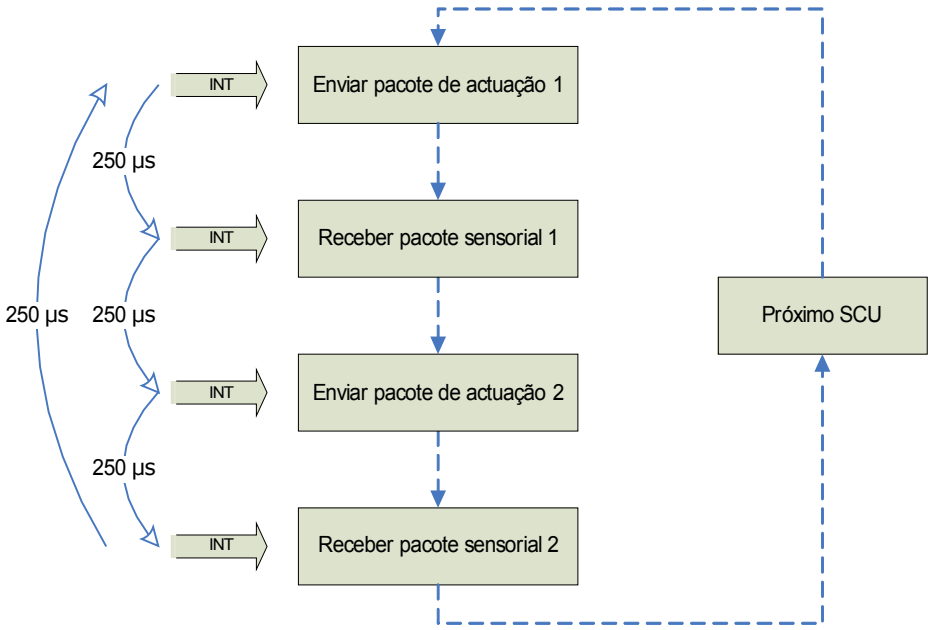


Fig. 4: Algoritmo de troca de informação pelo CAN no Master.

Módulo USART

Módulo com os *device drivers* para inicialização e implementação do protocolo USART.

Tabela 2: Funções para manipulação dos *buffers* da USART.

Funções	***Descrição***
usartInit	Inicialização da USART no PIC.
usartStreamMode	Esta função indica se o *buffer* de recepção está vazio ou não. Em caso negativo a USART está em modo de recepção de um *stream* de bytes (*true* retornado).
usartStoreRx	Armazenamento de um byte no *buffer* de recepção.
usartGetTx	Obtenção de um byte armazenado no *buffer* de transmissão.
usartResetRxBuff	Reinicialização a zero de todo o *buffer* de recepção.
usartResetTxBuff	Reinicialização a zero de todo o *buffer* de transmissão.
usartStore2extBuff	Armazenamento de um byte no *buffer* externo.

A Tabela 3 apresenta um conjunto de funções de mais alto nível, que fazem uso dos *device drivers* explícitos atrás, para processamento dos comandos provenientes do PC e construção da mensagem de resposta a retornar.

Tabela 3: Funções de construção da mensagem de resposta para uso da Rotina de Serviço à Interrupção.

Funções	***Descrição***
usartSendTestMsg	Rotina de envio de uma mensagem com o formato FA F9 F8 F7 F6 F5 (hex). Esta mensagem tem como propósito testar o funcionamento da USART.
usartSendStatus	No caso dos comandos de actuação, dois tipos de mensagens podem ser retornados, excluindo as mensagens com erros de recepção por parte do master. Elas são a confirmação da actuação com sucesso ou parâmetros inválidos. Em

Funções	*Descrição*
	tais casos é enviada uma mensagem com todos os bytes iguais ao do comando de actuação à excepção do primeiro com a indicação do estado: MESSAGE_SUCESS (0xFB) ou MESSAGE_INVREQ (0xFC). Esta função tem o objectivo de construir uma mensagem igual à última recebida com o primeiro byte igual ao estado a retornar.
usartProcessMsg	Rotina de processamento dos comandos enviados do PC para o master para construção da mensagem de resposta.

Módulo CAN

Módulo com a implementação das funções de alto nível para a recepção e o envio de mensagens CAN.

Tabela 4: Funções de alto nível para troca de mensagens via CAN.

Funções	*Descrição*
canInit	Inicialização do periférico CAN e dos *timers* e interrupções associados.
canSendMsg	Envio de uma mensagem CAN com dados de actuação a aplicar a um determinado SCU.
canReceiveMsg	Recepção de uma mensagem CAN com os dados sensoriais de um determinado SCU.
canClearStatus	Reinicialização do estado de erro do barramento CAN.

Módulo CANDRIVERS

Módulo com os device drivers básicos para inicialização e transmissão/recepção de pacotes via CAN.

Tabela 5: Device drivers da comunicação CAN.

Funções	*Descrição*
myCANInitialize	Inicialização do periférico CAN.
myCANSendMessage	Envio de um pacote através do primeiro *buffer* de transmissão vazio.
myCANSendMessage0	Envio de um pacote através do *buffer* de transmissão 0.
myCANSendMessage1	Envio de um pacote através do *buffer* de transmissão 1.
myCANSendMessage2	Envio de um pacote através do *buffer* de transmissão 2.
myCANReceiveMessage	Leitura de um pacote no primeiro *buffer* de recepção cheio.
myCANReceiveMessage0	Leitura de um pacote no *buffer* de recepção 0.
myCANReceiveMessage1	Leitura de um pacote no *buffer* de recepção 1.

Inicialização do módulo CAN do PIC:

1. Colocação do PIC no modo de configuração;
2. Definição da duração de cada segmento de bit;
3. Definição do *bitrate* para a velocidade pretendida;
4. Configuração da máscara de recepção;
5. Configuração dos dois filtros para cada um dos *buffers* de recepção;
6. Definição do formato *standard* para os identificadores dos pacotes a transmitir;
7. Definição de prioridades entre *buffers* de transmissão;
8. Configuração da recepção para apenas pacotes válidos com identificadores de 11 bits;
9. *Overflow* do *buffer* 0 para o *buffer* 1 desactivado;
10. *Clear* dos *interrupt flags* de recepção;
11. Abertura dos *buffers* para recepção;
12. Colocação do PIC no modo de operação normal.

Módulo TYPES

Módulo com a definição de tipos de variáveis extra para uso nos restantes módulos. Eles são:

Tipo de variável	***Descrição***
bool	Tipo booleano (*true* ou *false*) (tipo enumerado).
byte	Tipo inteiro de 8 bits sem sinal (*unsigned char*).
word	Tipo inteiro de 16 bits sem sinal (*unsigned int*).
dword	Tipo inteiro de 32 bits sem sinal (*unsigned long*).

Módulo P18F258

Biblioteca com a definição de todos os registos e bits correspondentes do PIC 18F258 para o seu controlo (ver *datasheet* da *Microchip*, PIC18F258).

3. Programa SLAVE

Localização: **<CD_PROJ>:\Lab\Fase3_Integration\Last Version\Slave_Stable_2.00**

A Fig. 5 apresenta as relações de inclusão entre os vários módulos de *software* da unidade slave.

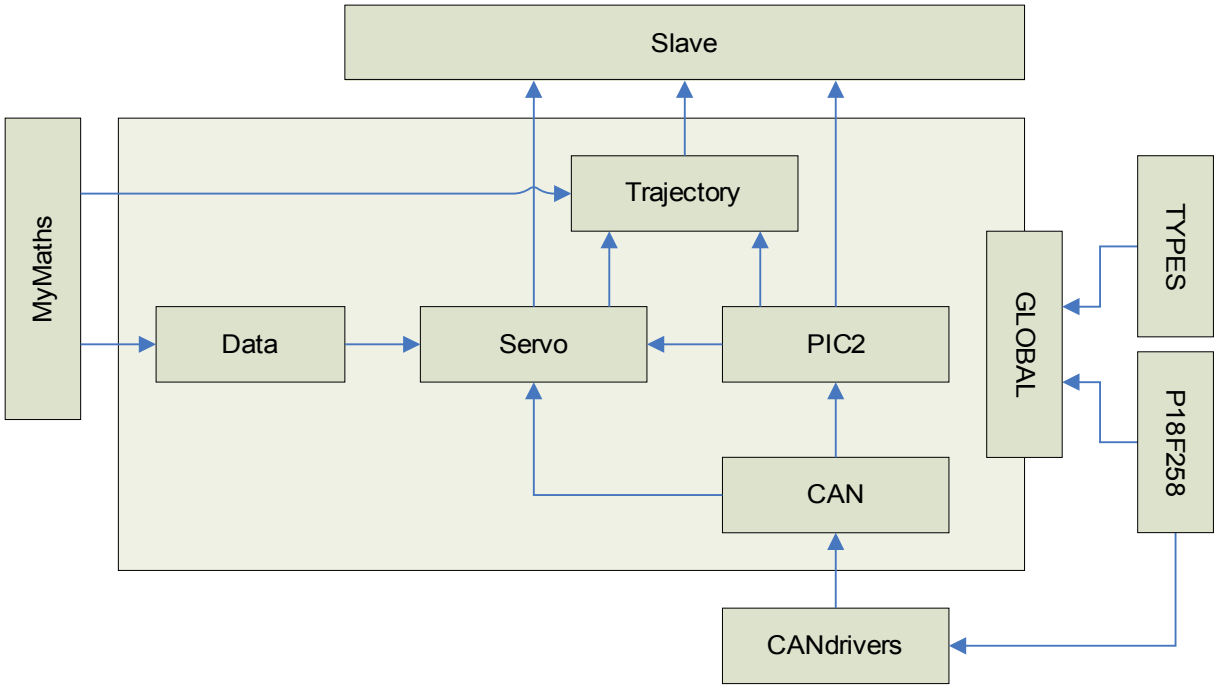


Fig. 5: Relações de inclusão dos módulos de software de cada Slave.

Módulo Slave

Programa principal com a invocação das rotinas de inicialização do PIC e de implementação dos controladores de equilíbrio e de posição/velocidade dos servomotores (Fig. 6).

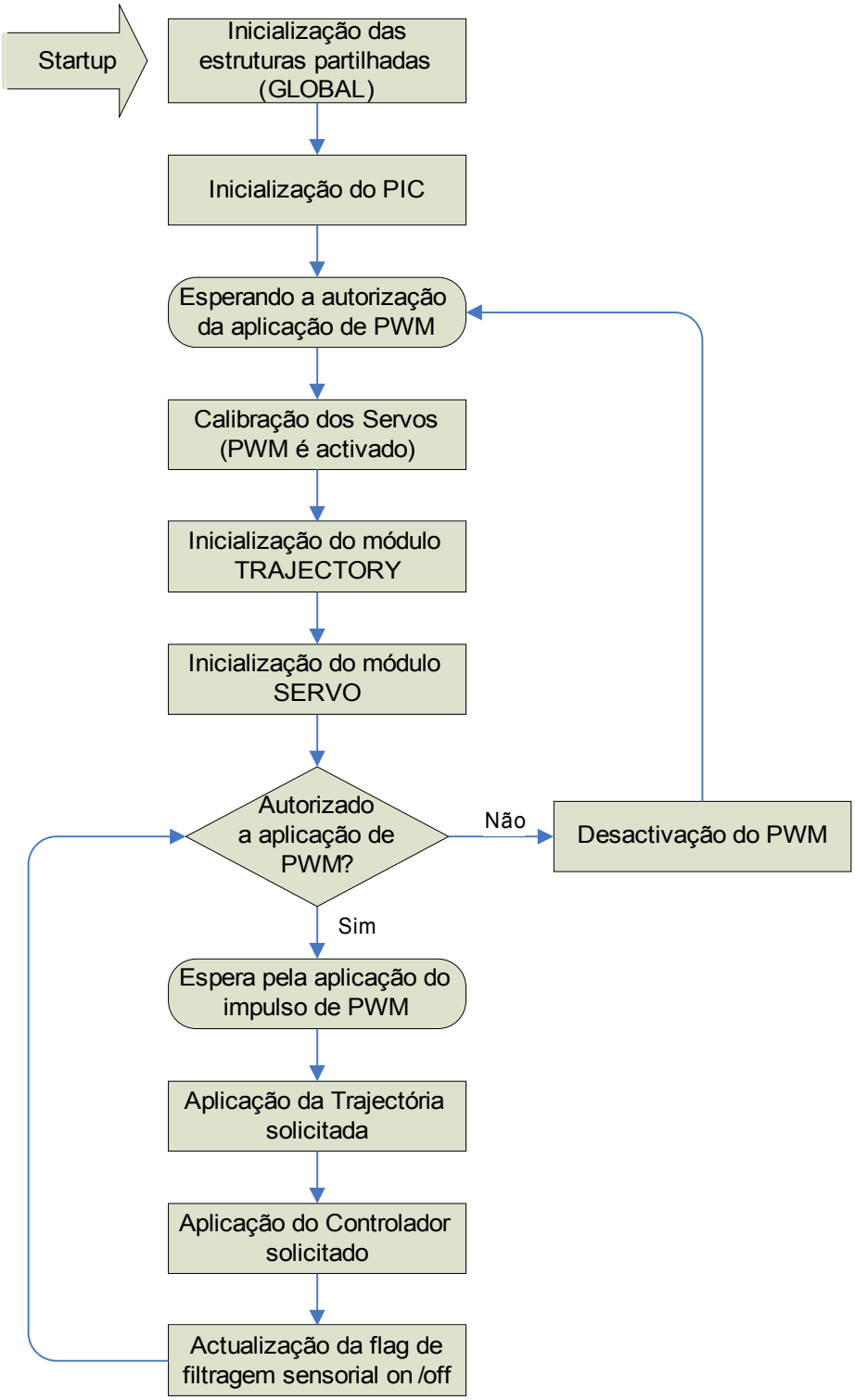


Fig. 6: Algoritmo da função *main* no programa principal *slave*.

Módulo TRAJECTORY

Este módulo implementa o controlo de velocidade nos servomotores, pela aplicação de posições segundo uma determinada trajectória, que imprimem a cada servomotor uma determinada velocidade média e eliminam as descontinuidades da forma de deltas de Dirac nas trajectórias de velocidade e de aceleração. A trajectória implementada segue a curva de um polinómio de terceiro grau.

Tabela 6: Funções globais da biblioteca *TRAJECTORY*.

Função	*Descrição*
initTrajectory	Inicialização dos parâmetros estáticos utilizados na realização de cada trajectória.
trajectory	Cálculo e realização iterativa de uma trajectória.

As variáveis indicadas na função *initTrajectory* são apresentadas a seguir:

```
// Estrutura com os dados necessários acerca de um trajecto
typedef struct {
    enum_trajectType type;      // Tipo de trajectória em execução
    double coef[4];             // Coeficientes da função theta=a0+a1*t+a2*t^2+a3*t^3
    word period;                // Duração em ticks da trajectória em execução
    word time;                  // Tempo em curso (em ticks)
    signed char theta_final;    // Posição final da trajectória (valor referência)
    byte control_type;          // Tipo do controlador em aplicação
} struct_trajectory;
static struct_trajectory traject[N_SERVOS];
```

Dos tipos de trajectória que podem ser realizados (*type*) discernem-se dois:

- *Free traject*: Nenhuma trajectória será aplicada, e a posição final *theta_final* a atingir é directamente atribuída ao servomotor em causa. Este tipo de trajecto é aplicado quando a duração solicitada (period) é nula.
- *Normal traject*: Realização da trajectória polinomial de coeficientes *coef*, cuja duração do trajecto é *period* e a posição final é *theta_final*. Esta trajectória é implementada sempre que *period* é positivo.

A Tabela 7 visualiza as rotinas invocadas pelo função global *trajectory*.

Tabela 7: Funções estáticas invocadas pela função *trajectory* da biblioteca *TRAJECTORY*.

Função	*Descrição*
executeTrajectory	Execução de uma de duas trajectórias: *free_traject* ou *normal_traject*.
calcTrajectory	Cálculo de uma nova trajectória a realizar.

A rotina *executeTrajectory* é sempre executada, implementando um dos dois tipos de trajecto apresentados, usando o parâmetro *theta_final* (posição final) no caso *free traject* (trajecto livre), e os parâmetros *time* e *coef* para o caso *normal_traject* (trajecto polinomial):

- *time* indica o tempo decorrido desde o início da trajectória;
- *coef* são os coeficientes do polinómio de terceira ordem $\{a_0, a_1, a_2, a_3\}$ utilizados para o cálculo da posição a aplicar para o instante *time*:

$$\theta = a_0 + a_1 \cdot time + a_2 \cdot time^2 + a_3 \cdot time^3$$

Sempre que a unidade Master solicita uma posição ou um tipo de controlador diferente de *theta_final* e/ou de *control_type* respectivamente, a trajectória em execução é interrompida, e um novo trajecto é calculado tendo em conta os novos valores referência. A rotina *calcTrajectory* realiza esta tarefa calculando uma nova trajectória de acordo com a duração solicitada pelo Master (*period*):

- Duração nula: execução do trajecto *free traject* indicando a opção através da variável *type*.
- Duração positiva: execução do trajecto polinomial (*normal traject*), calculando para isso os coeficientes *coef* do polinómio de terceiro grau a aplicar.

Em ambos os casos, as restantes variáveis estáticas da estrutura *struct_trajectory* são redefinidas tendo em conta o novo trajecto.

Módulo SERVO

Este módulo implementa as rotinas de compensação de posição dos servomotores e de equilíbrio com base nas forças de reacção. Outras rotinas são usadas para calibração ou para definição de parâmetros a usar pelos controladores (Tabela 8).

Tabela 8: Funções globais da biblioteca *SERVO*.

Função	***Descrição***
calibration	Calibração dos servomotores. É esperada a primeira ordem de actuação proveniente da unidade Master, e seguidamente é atribuído o PWM correspondente à posição solicitada a cada servomotor. Após o término do deslocamento, a calibração de medição da posição é efectuada com a associação da tensão à saída do potenciómetro à posição solicitada.
servoRequest	Atribuição da posição final a atingir. Ao contrário da função *servoActuation* do módulo *PIC2*, a posição solicitada não é atribuída imediatamente ao servo em causa, mas fica armazenada para utilização por parte dos controladores. Estes por sua vez, utilizarão este valor como a posição de referência a ser atingida através dos algoritmos implementados – PID no caso do controlo de posição.
initController	Inicialização das variáveis estáticas utilizadas pelos algoritmos de compensação. Apenas as variáveis associadas ao controlo de posição PID são inicializadas.
controller	Execução dos algoritmos de controlo.

A rotina de calibração (*calibration*) é executada sempre que o sistema arranca, e segue os seguintes algoritmos:

Para os sensores dos servomotores...

1. A primeira mensagem de actuação já chegou? Só passar para o passo 2, quando afirmativo;
2. Actuação sobre os servomotores de forma a cumprir a posição solicitada pela unidade *master*;
3. Esperar dois segundos para a cumprimento do movimento e estabilização do sinal de posição;
4. Amostragem de 25 medidas da saída do servo (25 períodos de PWM de duração);
5. Cálculo da média aritmética do valor medido;
6. A partir da seguinte equação (baseada na Equação 2) determinar o valor de *b* a partir da posição solicitada pelo master e do valor médio ADC_{res}:
$$b = pos_{master} + ADC_{media} * m$$
7. Activação dos filtros de medição sensorial.

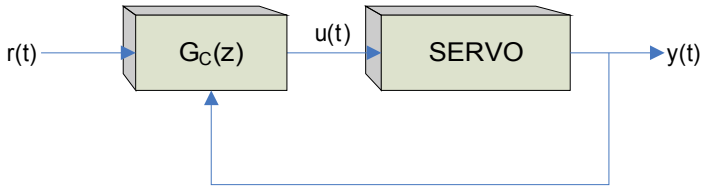
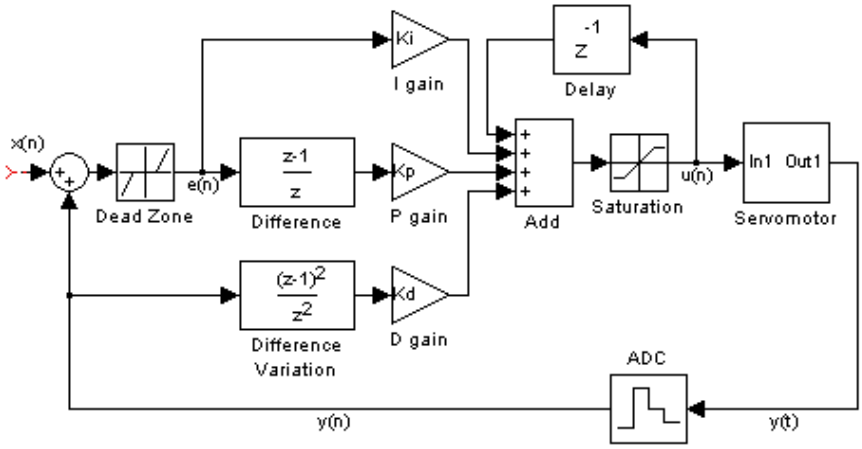
E para os sensores de força...

1. Leitura e armazenamento de 25 valores de cada sensor de força considerando um *offset* nulo;
2. Cálculo da média dos 25 valores ($Sensor_{original}$);
3. Armazenamento do novo valor de *offset* a usar nas medições consequentes:
$$offset(7\ bits) = 128\ (7\ bits) - Sensor_{original}(7\ bits)$$

Tabela 9: Tipo de controlo a seleccionar no campo PARAM_CONTROLON.

Tipo de Controlo sobre as juntas	*Designação*	*PARAM_CONTROLON*
Controlo em malha aberta	*NO_CONTROL*	*0b*00
Controlo de posição	*LOCOMOTION_CONTROL*	*0b*01
Controlo das forças de reacção	*REACTION_CONTROL*	*0b*10
Controlo de equilíbrio no tronco	*BALANCE_CONTROL*	*0b*11

Tabela 10: Controladores implementados.

Tipo de Controlador	*Descrição*
Controlo de posição dos Servomotores (LOCOMOTION_CONTROL)	 **Fig. 7: Modelo do controlo de posição dos servomotores.**  **Fig. 8: Compensador PID incremental para os Servomotores.**
Controlo de Equilíbrio segundo a lei proporcional (REACTION_CONTROL)	$\vec{v} = \vec{K}_P \cdot \vec{e} \quad \Leftrightarrow \quad \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} K_{Px} \\ K_{Py} \end{bmatrix} . \begin{bmatrix} e_x \\ e_y \end{bmatrix}$ $\vec{e} = Co\vec{P}_{ref} - Co\vec{P}_{med} \quad \Leftrightarrow \quad \begin{bmatrix} e_x \\ e_y \end{bmatrix} = \begin{bmatrix} CoP_{ref(x)} \\ CoP_{ref(y)} \end{bmatrix} - \begin{bmatrix} CoP_{med(x)} \\ CoP_{med(y)} \end{bmatrix}$ ● Junta lateral do pé: $v_x = K_{Px} \cdot e_x$ ● Junta dianteira do pé: $v_y = K_{Py} \cdot e_y$
Controlo de Equilíbrio usando a matriz Jacobiana (BALANCE_CONTROL)	$\vec{v} = \vec{K}_P \cdot \left[\vec{J}^T(q) \times \vec{e} \right] \quad \Leftrightarrow \quad \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} K_{P0} \\ K_{P1} \\ K_{P2} \end{bmatrix} \cdot \left(\begin{bmatrix} J_{11} & J_{12} & J_{13} \\ J_{21} & J_{22} & J_{23} \\ J_{31} & J_{32} & J_{33} \end{bmatrix}^T \times \begin{bmatrix} e_x \\ e_y \\ e_z \end{bmatrix} \right)$ ● Junta lateral do pé: $v_0 = K_{P0} \cdot [J_{11}(q) \cdot e_x + J_{21}(q) \cdot e_y]$ ● Junta dianteira do pé: $v_1 = K_{P1} \cdot [J_{12}(q) \cdot e_x + J_{22}(q) \cdot e_y]$ ● Junta do joelho: $v_2 = K_{P2} \cdot [J_{13}(q) \cdot e_x + J_{23}(q) \cdot e_y]$

Esta biblioteca implementa todos os algoritmos de controlo a executar na estrutura humanóide. A Tabela 9 apresenta os quatro tipos de controlo que podem ser seleccionados através de ordens de actuação executadas pela unidade de controlo principal – o PC – utilizando os barramentos de comunicações RS-232 e CAN. As rotinas que implementam cada um dos quatro algoritmos de controlo apresentam-se na Tabela 11 e são invocadas pela função *controller*.

Tabela 11: Rotinas de implementação da compensação chamadas pela função *controller*.

Função	***Descrição***
openloopControl	Controlo dos servomotores em malha aberta. A posição indicada na chamada da função *servoRequest* é atribuída ao servo em causa, sem passar por qualquer algoritmos de compensação.
locomotionControl	Compensação PID da posição de cada servomotor.
reactionControl	Compensação das forças de reacção aplicadas nos pés segundo o modelo proporcional.
JacobianControl	Compensação das forças de reacção através da matriz Jacobiana.

Como os algoritmos de controlo de equilíbrio (*reactionControl* e *jacobianControl*) baseiam-se no controlo de velocidade e não directamente pela posição, construiu-se a rotina *applyVelocity* que implementa a acção de velocidade através do in/decremento de posição. Adicionalmente foi incorporado um limitador de velocidade de modo a evitar qualquer tipo de instabilidade na acção do controlador.

Módulo DATA

Este módulo contém as ferramentas necessárias para a implementação dos algoritmos de equilíbrio presentes na biblioteca *SERVO*. Além disso possui todas as características físicas do pé e da perna utilizados (Tabela 12 e Tabela 13).

Tabela 12: Características do pé sensível às forças de reacção.

	Localização de cada sensor (mm)	
Sensor	*Coordenada x*	*Coordenada y*
Dianteiro esquerdo	-32.5	6.20
Dianteiro direito	32.0	6.20
Traseiro esquerdo	-35.0	-6.25
Traseiro direito	34.0	-6.25

Tabela 13: Características físicas da perna.

Parâmetro	***Símbolo***	***Valor***
Massa do elo inferior	M_1	450 *g*
Massa do elo superior	M_2	450 *g*
Comprimento do elo superior	L_1	95 *mm*
Comprimento do elo inferior	L_2	95 *mm*
CoM do elo superior	R_1	47 *mm*
CoM do elo superior	R_2	47 *mm*

A localização dos sensores de força (Tabela 12) foram obtidas por medição directa, enquanto que as características da perna foram consultadas no relatório do ano lectivo anterior (Tabela 13).

Tabela 14: Funções do módulo *DATA*.

Função	***Descrição***
calcCoP	Cálculo das coordenadas do centro de pressão com base nas forças de reacção.
jacobianMatrix	Cálculo da matriz Jacobiana para aplicação do controlador de equilíbrio baseado no Jacobiano.

Para o cálculo do centro de pressão são relacionados os parâmetros de localização dos sensores de força indicados na tabela Tabela 12 com as suas saídas.
No que toca à matriz Jacobiana as grandezas da Tabela 13 são utilizadas para o cálculo de cada elemento da matriz usando também as funções trigonométricas definidas na biblioteca *mymaths*.

Módulo MyMATHS

Esta biblioteca implementa as funções matemáticas necessárias para a programação dos módulos *SERVO* e *TRAJECTORY*. As funções *seno* e *coseno* apenas serão úteis para a implementação do controlador das forças usando a matriz Jacobiana.

Função	***Descrição***
abs	Cálculo do módulo de um valor inteiro com sinal *signed int*. Retorno no formato *unsigned int*.
seno	Cálculo da função trigonométrica seno. O argumento é passado em graus e o retorno é um valor inteiro com sinal *signed char* entre -100 e +100: $retorno = resultado \times 100$
coseno	Cálculo da função trigonométrica coseno. O argumento é passado em graus e o retorno é um valor inteiro com sinal *signed char* entre -100 e +100.

As funções seno e coseno são implementadas recorrendo a uma *lookup table* com a associação do resultado a cada valor angular (Tabela 15).

Tabela 15: *Lookup table* para a função seno.

Ângulo **(º)**	***Seno*** **(×100)**
0	0
1	1
2	3
3	5
4	6
5	8
...	...
81	98
82	99
83	99
...	...
89	99
90	100

Apenas nos é útil armazenar a gama de 0 a +90º, uma vez que o conjunto de resultados repete-se para outros ângulos fora deste intervalo. Deve-se, contudo, adaptar o argumento para a gama [0,90]º e aplicar correctamente o sinal ao resultado final.
No que respeita à resolução da *lookup table*, é suficiente armazenar o resultado para cada ângulo inteiro, uma vez que o algoritmos de medição sensorial do potenciómetro de posição de cada servomotor também só consegue medir com uma resolução de 1º.

De notar que a *lookup table* da função seno (Tabela 15) também pode ser utilizada para o cálculo do coseno:

$$coseno(\theta) = seno(90º - \theta)$$

... pelo que apenas uma *lookup table* é suficiente.

Módulo PIC2

Este módulo define as funções de controlo de baixo nível dos servomotores a serem utilizadas pelas bibliotecas de mais alto nível, e as rotinas de atendimento às interrupções responsáveis pela gestão do PWM de actuação e pela leitura sensorial dos servos.

Tabela 16: Funções de acesso externo do módulo *PIC2*.

Função	***Descrição***
initPic	Inicializações relativas às configurações dos periféricos do microcontrolador.
wait	Função bloqueante que gera um atraso de *n* ms (*n* passado como argumento). Esta função faz uso dos *timers* relativos à actuação.
waitTick	Função bloqueante que espera pelo período de PWM seguinte (20ms no pior dos casos).
servoActuation	Actuação directa sobre a posição dos servomotores. As variáveis com a informação do número de iterações a manter o sinal de PWM a 1 durante a zona de descida são actualizadas para as posições solicitadas.
statusPWM	Activação/desactivação dos sinais de PWM à saída do PIC.
statusFilter	Activação/desactivação dos filtros aplicados à posição medida.
limitPosition	Limitação do um valor entre os extremos de posição do servo: -90 e +90º.

A função que mais se destaca desta lista, é sem dúvida, a *servoActuation* pois é ela que define qual deve ser ao *duty-cycle* do sinal de PWM. É esta função que inicializa as variáveis de duração do impulso de PWM que posteriormente serão comparadas com um contador durante a zona de descida de PWM.

Tabela 17: Funções internas do módulo *PIC2*.

Função	***Descrição***
initLocal	Inicialização das estruturas de dados locais ao módulo.
sampleExtraSensors	Leitura dos sensores de força.
updateServoMeasures	Medição iterativa do sinal de saída do servo ao longo de um período de PWM, para cálculo da tensão mínima e da largura do impulso de corrente (Fig. 16).

Função	***Descrição***
finalizeServoMeasures	Finalização do processamento sensorial (executado no fim do período de PWM). A posição angular do servo (em graus) e a corrente consumida normalizada entre 0 e 100 são determinados (Fig. 17).
filter	Filtragem da posição medida usando métodos lineares e não-lineares.
delay	Geração de um atraso recorrendo somente à instrução *nop* (*no operation*).
highISR	Rotina de serviço à interrupção de alta prioridade. Nesta rotina é feita gestão da actuação e do processamento sensorial dos servomotores.
lowISR	Rotina de serviço à interrupção de baixa prioridade. Nesta rotina é feita a gestão das comunicações CAN da unidade *slave*.

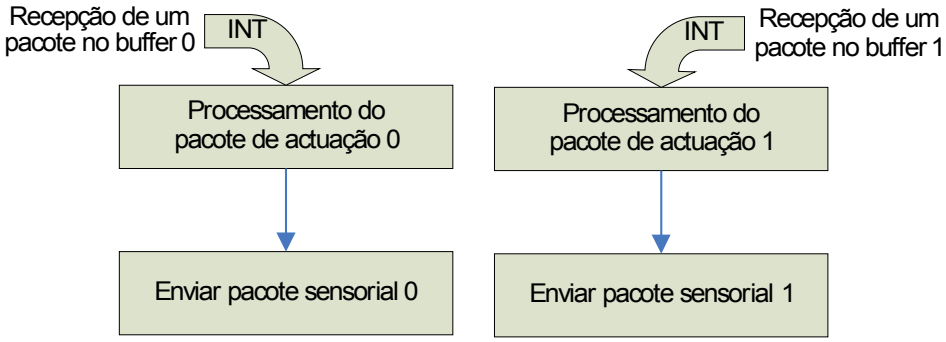


Fig. 9: Algoritmo de troca de informação pelo CAN no Slave (*lowISR*).

A Fig. 10 e a Fig. 11 descrevem a organização temporal das interrupções atendidas na rotina de serviço às altas prioridades, no que respeita à actuação e leitura sensorial respectivamente.

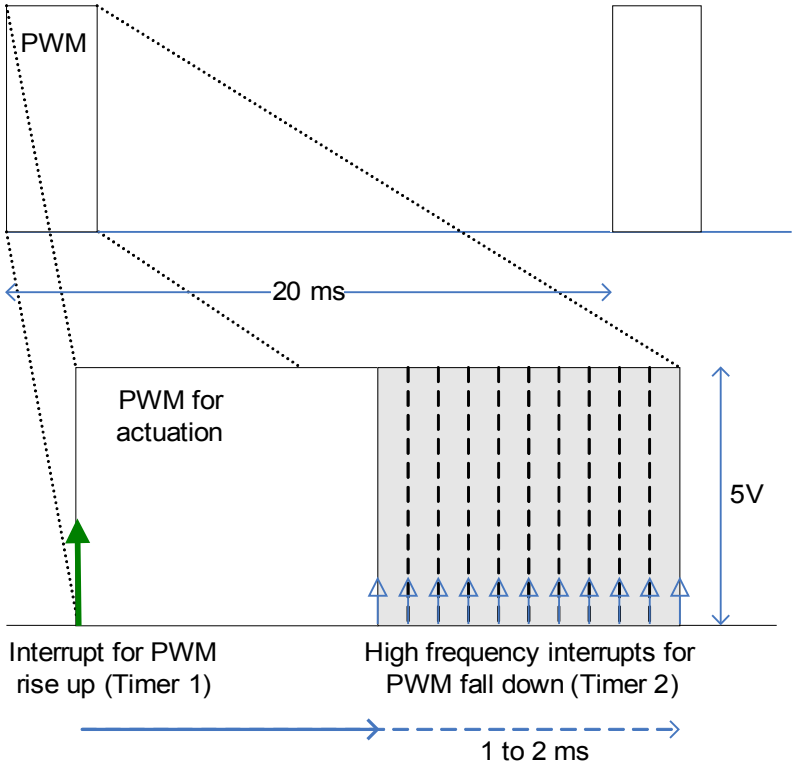


Fig. 10: Interrupções na geração do PWM de actuação.

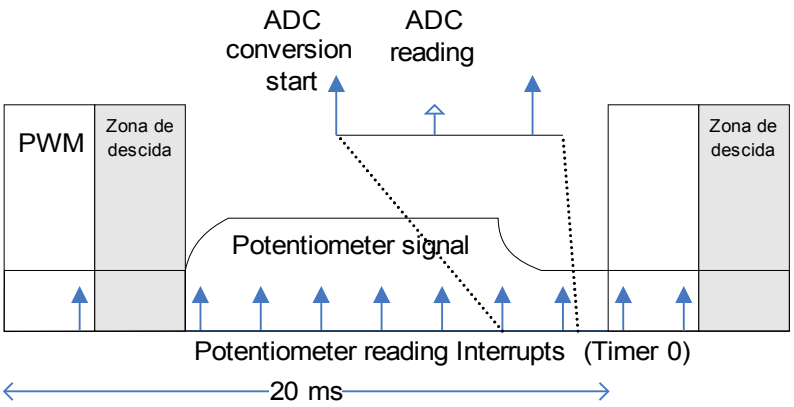


Fig. 11: Interrupções na medição sensorial.

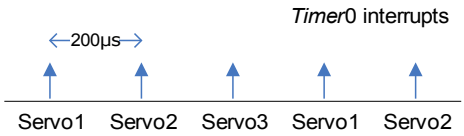


Fig. 12: Multiplexagem dos servos.

A Fig. 13 apresenta o diagrama de blocos do algoritmo em funcionamento na rotina de atendimento às altas interrupções.

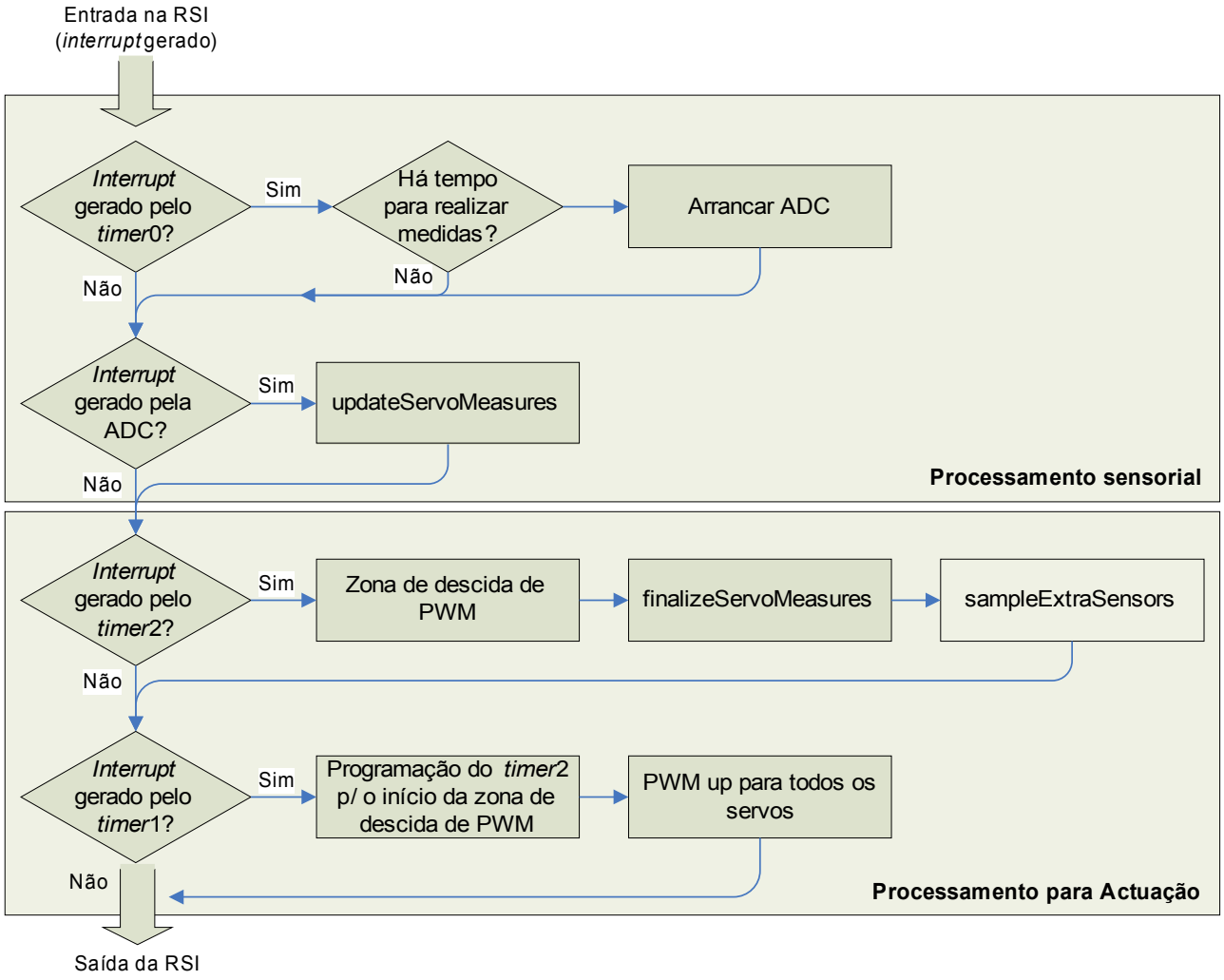


Fig. 13: Algoritmo da RSI de alta prioridade.

As figuras seguintes entram em mais detalhe nas acções de actuação e de leitura sensorial ao longo de cada período de PWM.

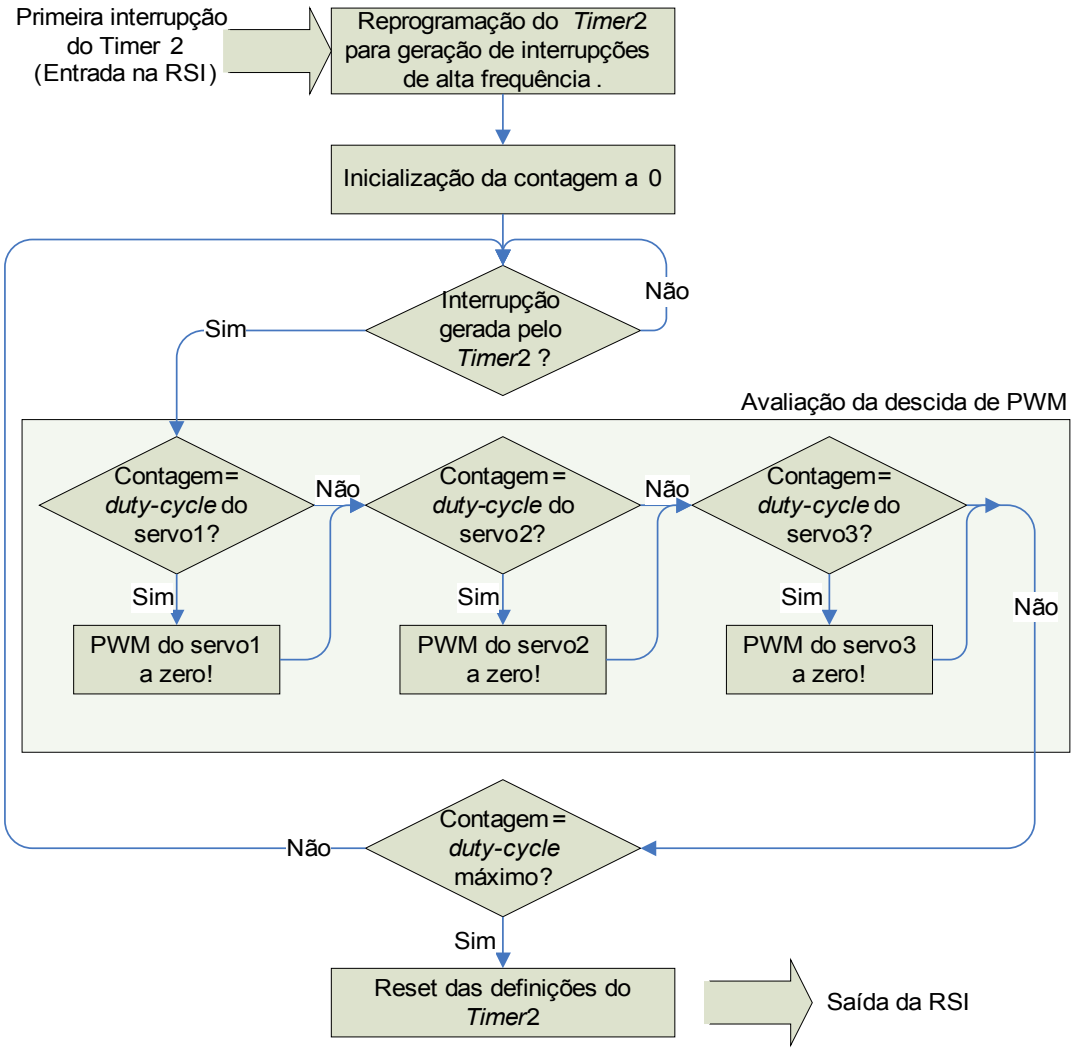


Fig. 14: Atendimento às interrupções de alta frequência (zona de descida do PWM).

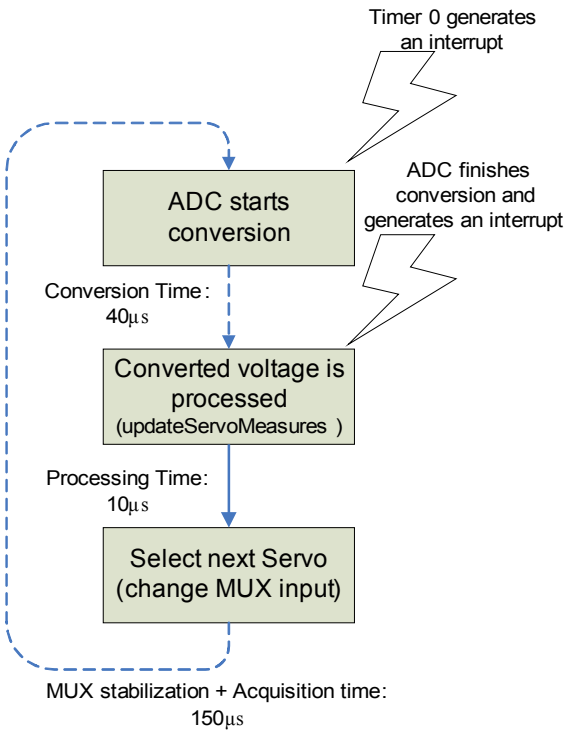


Fig. 15: Algoritmo de leitura dos três servomotores.

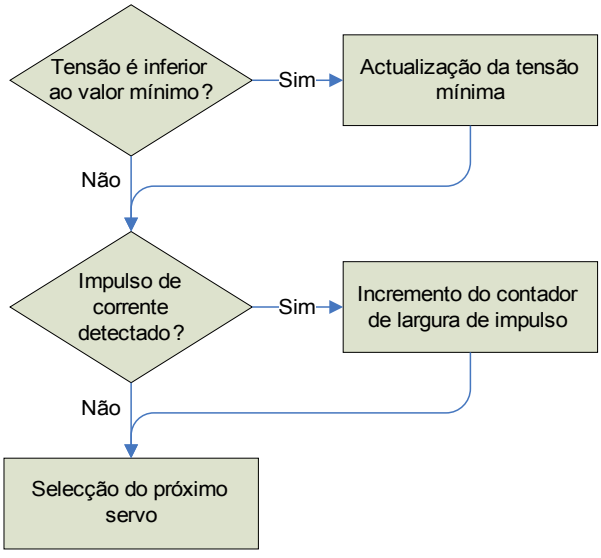


Fig. 16: *updateServoMeasures*

No fim da aplicação de cada impulso de PWM, as medidas efectuadas ao longo do período, são consideradas para determinação dos valores definitivos (*finalizeServoMeasures* e *sampleExtraSensors*).

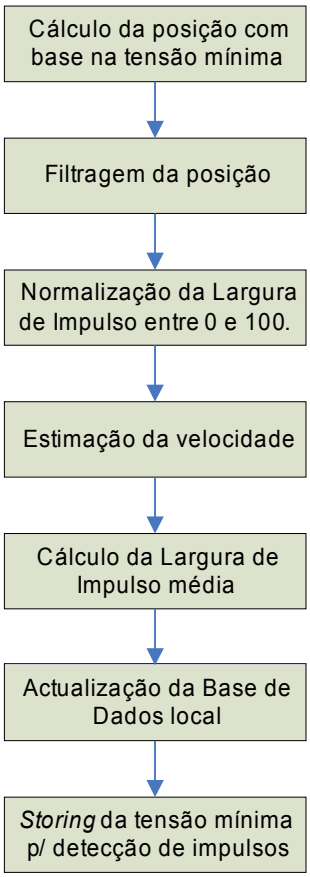


Fig. 17:
finalizeServoMeasures

Algoritmo de amostragem dos sensores de força (*sampleExtraSensors*) no fim da aplicação do impulso de PWM:

1. Selecção da entrada do multiplexer associado ao sensor a ler;
2. Espera de 200ns para estabilização do sinal à saída do multiplexer e à entrada da ADC;
3. Arranque da ADC e espera até a conversão finalizar;
4. **Leitura do valor convertido pelo algoritmo indicado previamente;**
5. Todos os sensores lidos? Em caso negativo passar ao sensor seguinte e voltar ao passo 1;
6. Em caso afirmativo, seleccionar a entrada do multiplexer associado ao primeiro servomotor a ler;
7. Deixar estabilizar os sinais (200ns).

Algoritmo de leitura de um sensor de força:

1. Amostragem do valor analógico na representação de 8 bits (*char*) – excursão de 0 a 255;
2. Redução da resolução para metade (7 bits) – excursão entre 0 e 127;
3. Correcção do resultado pela adição do *offset* determinado durante a calibração:

$$Resultado(8\ bits) = Sensor_{output}(7\ bits) + offset(7\ bits)$$

Módulo CAN

Módulo com a implementação das funções de alto nível para a recepção e o envio de mensagens CAN, respeitantes à unidade slave.

Tabela 18: Funções de alto nível para troca de mensagens via CAN.

Funções	***Descrição***
initCan	Inicialização do periférico CAN e das interrupções associadas.
sendCanMessage	Envio de uma mensagem CAN com dados de sensoriais para a unidade master.
receiveCanMessage	Recepção de uma mensagem CAN proveniente do master com os dados de actuação a aplicar.
checkCan	Verificação do bloqueio do CAN, no caso do PIC se encontrar no modo *bus-off*.

A função *checkCan* é útil para verificar se a unidade ainda possui a interface CAN activa e em funcionamento. Cada nó na rede pode comportar-se de três formas diferentes durante o seu funcionamento:

- Modo *error-active*: mensagens normais e frames de erro (na ocorrência de erros) com bits dominantes são trocadas com os outros nós para lhes indicar a ocorrência de anomalias;
- Modo *error-passive*: as frames de erro passam a ser constituídas por bits recessivos para evitar a interferência destrutivas das mensagens provenientes de outros nós;
- Modo *bus-off*: o nó é bloqueado em termos de recepções e transmissões.

Na ocorrência de erros, um contador é incrementado, sendo também decrementado na ausência deles. Quando atinge um valor limite o modo de funcionamento vai alternando para o modo seguinte até atingir o modo *bus-off*, bloqueando as recepções e as transmissões com os outros nós.
Esta função foi construída devido ao facto de as unidades esporadicamente bloquearem as suas comunicações e ter assim um meio para verificar a causa.

Módulo CANDRIVERS

(Igual ao do programa Master – capítulo 2)

Módulo GLOBAL

Módulo com a definição da base de dados com os dados de actuação e sensorial do próprio SCU.

Base de dados sensorial

```
// Estrutura descritiva dos sensores
typedef struct {
    struct {
        bool pwm;                                   // Motores ligados/desligados
        bool filter;                                // Filtros ligados/desligados
        bool deadlineError;                         // Violação de deadline
        bool motionFin[N_SERVOS];                   // Movimento terminado
    } sysStatus;                                  // Estado sensorial do sistema
    struct_servo servo[N_SERVOS];                 // Sensores dos servos (posição,
velocidade e corrente)
    unsigned char special[N_SPECIAL_SENSORS];     // Sensores especiais
} struct_sensors;

extern volatile struct_sensors   sensors;         // Sensores
```

Base de dados de actuação

```
// Estrutura descritiva dos actuadores
typedef struct {
    // Estrutura de actuação de status para cada SCU
    struct {
        bool pwm;                                     // PWM on/off
        bool filter;                                  // Filter on/off
    } sysStatus;                                  // Estado da actuação do sistema
    struct {
        byte kp, kd, ki;                                  // Parâmetros Kp, Kd e Ki
        bool type;                                        // Tipo de Controlador
    } control[N_SERVOS];                          // Controlador
    struct_servo servo[N_SERVOS];                 // Informação de actuação (posição,
velocidade e corrente)
} struct_actuators;

extern volatile struct_actuators actuators;       // Actuadores
```

Tabela 19: Funções presentes no módulo GLOBAL.

Funções	*Descrição*
initGlobal	Inicialização de toda a base de dados para os valores origem.
initSensors	Inicialização somente da base de dados sensorial.
initActuators	Inicialização somente da base de dados de actuação.
reinitServo	Reinicialização das posições dos actuadores para os valores origem (definidos à priori pelo próprio SCU).

Módulo TYPES

(Igual ao do programa Master – capítulo 2)

Módulo P18F258

(Igual ao do programa Master – capítulo 2)

4. Programas MATLAB

4.1. Device-Drivers para comunicação

Localização: **<CD_PROJ>:\Lab\Fase3_Integration\PC\Control\Motion**

Tabela 20: Lista de device drivers da unidade principal.

Funções M	*Função*	*Descrição*
initcom	`[H,state]=initcom(gate,rate)`	Criação de uma nova ligação.
killcom	`stat=killcom(H)`	Término da ligação.
calibcom	`calibcom(H)`	Pedido de envio de uma sequência de teste.
readcanstat	`[array,...]=readcanstat(H)`	Consulta do estado do barramento CAN.
readjoint	`[servos,...]=readjoint(H,scu_id,param)`	Leitura das posições das juntas de um SCU.
readspecial	`[special,...]=readspecial(H,scu_id)`	Leitura dos sensores especiais.
applyjoint	`[...]=applyjoint(H,scu_id,param,servos)`	Actuação nas juntas de um determinado SCU.
applycontrol	`[...]=applycontrol(H,scu_id,param,servos)`	Actualização dos parâmetros PID de uma determinado SCU.

initcom

```
Estabelecimento de uma nova ligação via RS-232

[handler,state]=initcom(gate,rate)

Entradas:
 gate    -> Porta a utilizar (1,2,...)
 rate    -> baudrate a definir
Saídas:
 handler -> ID da linha de comunicações
 state   -> Configurações da linha
```

killcom

```
Término de uma ligação RS-232 existente.

stat=killcom(handler)

Entradas:
 handler -> ID da linha série
Saídas:
 stat    -> retorna 1 em caso de sucesso
```

calibcom

```
Pedido de envio de uma sequência de teste por parte do master.

calibcom(handler)

Entradas:
 handler -> ID da linha série.
```

readcanstat

```
Leitura do estado do barramento CAN entre slaves.

[array,state,rx,error,errorstr,tries]=readcanstat(H)

Entradas:
 H => Handler para comunicar com o Master
Saídas:
 array    => [estado de erro, #erros de transmissão, #erros de recepção]
 state    => Bits de estado dos servos
 rx       => Mensagem de baixo nível recebida
 error    => Código de erro, se existente
 errorstr => String descritiva do erro
 tries    => Número de tentativas para efectuar a comunicação
```

readjoint

```
Leitura de um parametro sensorial dos servos de um SCU

[servos,state,rx,error,errorstr,tries]=readjoint(H,scu_id,param)

Entradas:
 H        => Handler para comunicar com o Master
 scu_id   => Identificador do SCU alvo
 param    => Parametro a ler (0:posição, 1:velocidade, 2:corrente)
Saídas:
 servos   => Parametro de saída [servo1,servo2,servo3]
 state    => Bits de estado dos servos
 rx       => Mensagem de baixo nível recebida
 error    => Código de erro, se existente
 errorstr => String descritiva do erro
 tries    => Número de tentativas para efectuar a comunicação
```

readspecial

```
Leitura dos sensores especiais (sensores de força ou giroscópio+inclinómetro).

[special,rx,error,errorstr,tries]=readspecial(H,scu_id)

Entradas:
 H    => Handler das comunicações com o Master
 scu_id   => Identificador do SCU alvo
Saídas:
 special  => Valores dos sensores especiais
 rx       => Mensagem de baixo nível recebida
 error    => Código de erro, se existente
 errorstr => String descritiva do erro
 tries    => Número de tentativas para efectuar a comunicação
```

applyjoint

```
Aplicação de uma ordem de posição ou velocidade a cada motor de uma junta.

[rx,error,errorstr,tries]=applyjoint(H,scu_id,param,servos)

Entradas:
 H        => Handler para comunicar com o Master
 scu_id   => Identificador do SCU alvo
 param    => Parametro a aplicar (0:posição, 1:velocidade)
 servos   => Dados a aplicar [servo1,servo2,servo3]
Saídas:
 rx       => Mensagem de baixo nível recebida
 error    => Código de erro, se existente
 errorstr => String descritiva do erro
 tries    => Número de tentativas para efectuar a comunicação
```

applycontrol

```
Ajuste dos parametros do controlador PID para o posicionamento do servo

[rx,error,errorstr,tries]=applycontrol(H,scu_id,param,servos)

Entradas:
 H        => Handler para comunicar com o Master
 scu_id   => Identificador do SCU alvo
 param    => Parametro a modificar (0:K_I, 1:K_P, 2:K_D, 3:PID on)
 servos   => Dados a aplicar [servo1,servo2,servo3]
Saídas:
 rx       => Mensagem de baixo nível recebida
 error    => Código de erro, se existente
 errorstr => String descritiva do erro
 tries    => Número de tentativas para efectuar a comunicação
```

Todos os *device drivers*, tanto os de consulta sensorial como os de actuação, seguem um algoritmo semelhante ao enunciado na Fig. 18.

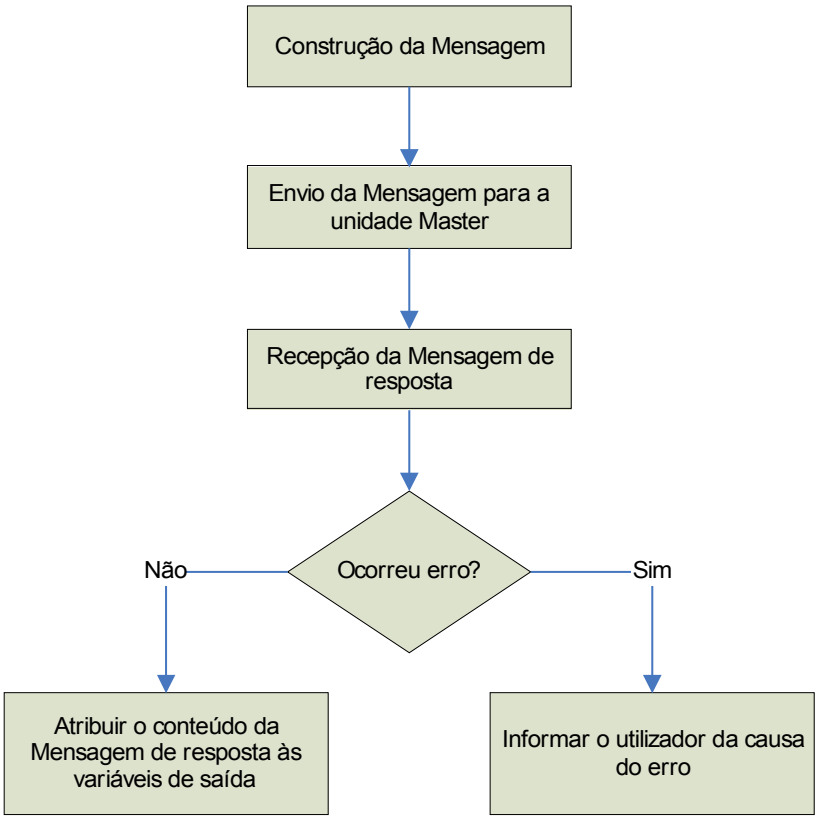


Fig. 18: Algoritmo geral dos *device drivers* da unidade principal.

A construção da mensagem baseia-se na definição de um *array* de bytes, cuja estrutura segue o formato de um comando PC→Master.

O envio da mensagem, constituído pelos vários bytes, é feita segundo o algoritmo descrito na Fig. 19. É enviado um byte de cada vez para a porta série, e caso ocorra algum erro são reinicializadas as comunicações voltando de seguida a tentar reenviar o mesmo byte. O processo só termina quando todos os bytes forem enviados com sucesso.

A recepção da mensagem de resposta está descrita na Fig. 20 e é executada logo após o envio do comando. A mini-toolbox *cport* oferece-nos já uma função que permite ler imediatamente um array de 6 bytes da porta série indicando se a recepção teve sucesso em cada um deles. Caso tal não aconteça, as comunicações são reinicializadas e o reenvio do comando é repetido com a consequente leitura da resposta.

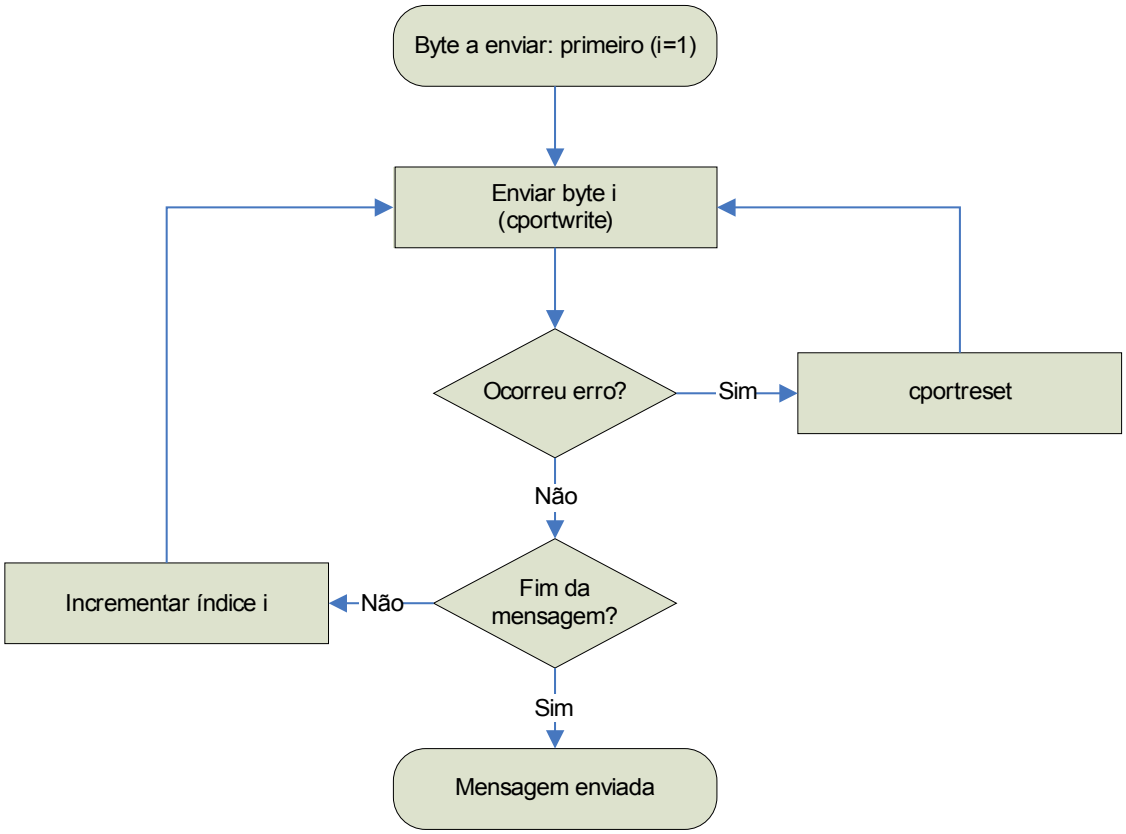


Fig. 19: Algoritmo de envio de uma mensagem para o Master.

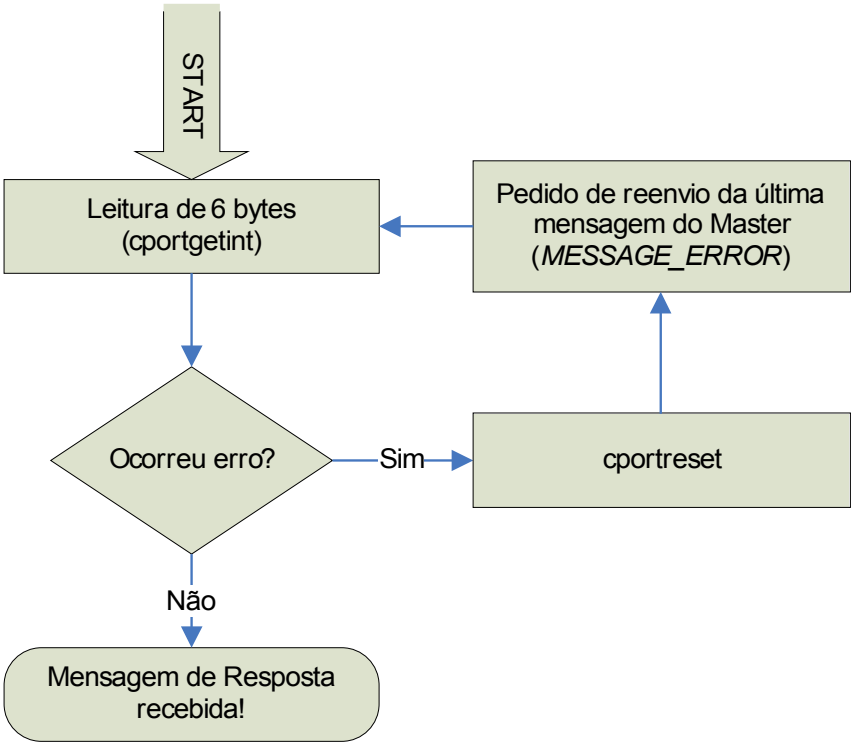


Fig. 20: Algoritmo para recepção da mensagem de resposta do Master.

4.2. Funções de Alto-Nível para Controlo do Robot Humanóide

a) Controlo de uma só Unidade de Controlo

Localização: **<CD_PROJ>:\Lab\Fase3_Integration\PC\Control\Motion**

Rotinas especiais foram escritas[1] para amostragem de trajectórias de uma só unidade de controlo.

Tabela 21: Lista de rotinas para amostragem de trajectórias de uma unidade de controlo.

Funções M	***Chamada da função***
predict_trajectory	`[posT,velT,accT]=predict_trajectory(pos0,posF,n_samples,ratio)` Previsão da trajectória ideal. A partir da posição inicial (*pos0*) e final (*posF*), mais o número total de amostras (*n_samples*) e a fracção correspondente à trajectória (*ratio*), é devolvida uma trajectória polinomial de terceiro grau em posição (*posT*), velocidade (*velT*) e aceleração (*acctT*).

sample_traj	`array=sample_traj(H,T,pos0,posF,k)`
	
	Amostragem da trajectória de um servo de uma unidade slave. A partir das informações da duração do movimento *T*, da posição inicial (*pos0*) e final (*posF*), e dos parâmetros k=[K_I,K_P,K_D], o primeiro servo da unidade slave de teste (endereço fornecido no interior da rotina – *scu*), um array (*array*) é produzido com a informação de tempo, posição, velocidade, corrente e as trajectórias previstas de posição e velocidade, dispostas em vectores coluna.
	
	T: Duração do Movimento
	pos0 e *posF*: Posição inicial e final
	$k = [K_I, K_P, K_D]$: Parâmetros de controlo PID)
	
	`array=[tempo posição velocidade corrente posT velT]`
	Todos os elementos são vectores coluna.

sample_traj3	`array=sample_traj3(H,T,pos0,posF,k)`
	
	Equivalente à rotina *sample_traj*, mas com a diferença de efectuar amostragem de todos os três servomotores de uma determinada unidade slave (introduzida no conteúdo – *scu*). *Pos0* e *posF* são vectores linha de três elementos, e *k* é uma matriz de três colunas para os três tipos de parâmetros K_I, K_P e K_D, por três linhas, correspondentes aos três servomotores.
	
	T: Duração do Movimento
	$pos0 \Leftrightarrow posF = [servo_1, servo_2, servo_3]$ $\quad k = \begin{bmatrix} K_{1I} & K_{1P} & K_{1D} \\ K_{2I} & K_{2P} & K_{2D} \\ K_{3I} & K_{3P} & K_{3D} \end{bmatrix} \begin{matrix} (servo_1) \\ (servo_2) \\ (servo_3) \end{matrix}$
	`array=[tempo` **`posição posT velocidade velT corrente`**`]`
	Todos os elementos à excepção do tempo (vector coluna), são matrizes com três colunas correspondentes aos três servomotores.

1 <CD_PROJ>:\Lab\Fase3_Integration\PC\Control\Motion\

b) Execução de Movimentos Coordenados entre várias Unidades

Adicionalmente construíram-se rotinas que permitissem a realização de movimentos coordenados entre várias unidades de controlo. A função *exe_traj* realiza essa tarefa, utilizando os *setpoints gerados* pela rotinas de planeamento de trajectórias da Tabela 22. Para a realização de outros movimentos apenas é necessário construir mais rotinas tendo em vista a geração dos *setpoints* apropriados às trajectórias em mente.

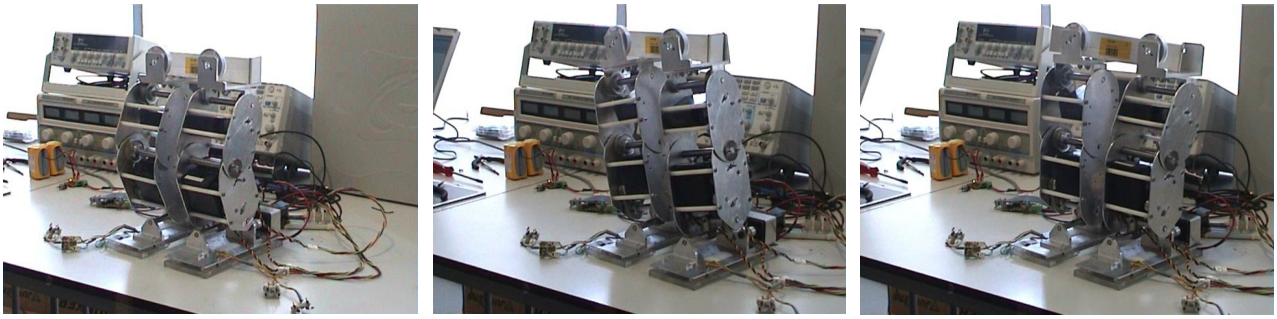


Fig. 21: Tipos de movimentos das pernas.

Tabela 22: Funções de cálculo das trajectórias para os diferentes tipos de movimentos.

Funções M	***Descrição***
flexao	`[pos_apply,k,t]=flexao(T,motion)` Com base no período de deslocamento *T* e o ângulo máximo de dobra do tornozelo *motion*, são calculados os parâmetros para execução do movimento de flexão das duas pernas.
inclinar	`[pos_apply,k,t]=inclinar(T,motion)` Cálculo dos parâmetros associados ao movimento lateral das pernas. Além do período de deslocamento *T* é fornecido o ângulo máximo de variação da junta lateral do pé *motion*.
levantar_perna	`[pos_apply,k,t]=levantar_perna(T,motion,perna)` Levantamento da perna até um ângulo máximo do joelho *motion* da perna indicada na variável *perna*. Se *perna* possuir um valor inválido (diferente de 1 e de 2), as duas efectuarão o movimento.

Os *setpoints* da trajectória calculada por cada uma destas funções são armazenadas em três variáveis:

- *pos_apply* : Setpoints de posição (final);
- *k* : parâmetros de compensação a adoptar;
- *t* : Periodo de deslocamento.

As características destas variáveis estão descritas na Tabela 23 onde...

- N_SCU é o número total de unidades slave: 8;
- N_SERVOS é o número de servos para cada unidade: 3
- N_MOTIONS é a quantidade de movimentos: normalmente 1.

Tabela 23: Descrição da estrutura das variáveis retornadas pelas funções de cálculo de trajectórias.

Variável	*Dimensão*	*Descrição*
pos_apply	N_SCU×N_SERVOS×N_MOTIONS (3 dimensões)	Posições finais para cada unidade, servo e movimento.
k	N_SCU×N_SERVOS×N_MOTIONS×4 (4 dimensões)	Parâmetros de compensação (4) para cada unidade, servo e movimento.
t	N_MOTIONS (1 dimensão)	Período de deslocamento para cada movimento.

Como funções para a execução efectiva dos deslocamentos temos os indicados na Tabela 24.

Tabela 24: Rotinas de execução de movimentos.

Funções M	*Descrição*
exe_traj	`[tt,position,velocity,current]=exe_traj(H,t,pos_apply,Textra)` Rotina de execução dos *setpoints* calculados pelas rotinas da Tabela 6.
recycle	`recycle(H,speed,correct_flag)` "Reciclagem" das juntas das pernas, reposicionado-as na posição vertical. *Speed* indica a velocidade a que deve ser feita a reciclagem (em ciclos de 20 ms) e *correct_flag* está relacionada com a correcção das posições standard, de modo a acertar a postura vertical das pernas.

Tabela 25: Parâmetros de retorno da função *exe_traj*.

Variável de retorno	*Dimensão*	*Descrição*
time	Vector coluna (1 dimensão)	Vector com o intervalo de tempo do movimento.
position	N_TEST×N_SERVOS×TIME (3 dimensões)	Posições medidas para cada unidade e cada servo ao longo do tempo.
velocity	N_TEST×N_SERVOS×TIME (3 dimensões)	Velocidade média medida para cada unidade e cada servo ao longo do tempo.
current	N_TEST×N_SERVOS×TIME (3 dimensões)	Corrente média medida para cada unidade e cada servo ao longo do tempo.

Outras funções adicionais utilizadas pelas rotinas da Tabela 22 e da Tabela 24 são apresentadas na Tabela 25. Embora não sejam de uso directo pelo utilizador, a sua edição pode revelar-se bastante útil para a correcção de pequenos desvios que podem verificar-se durante a execução das trajectórias.

Tabela 26: Funções adicionais utilizadas pelas rotinas de cálculo e execução de trajectórias.

Funções M	***Descrição***
controller	Definição e activação dos controladores PID a todas as unidades de controlo.
controller_off	Desactivação dos controladores PID em todas as unidades de controlo.
adaptate	`pos_final=adaptate(scuid,position)` Considerando que as pernas na postura vertical apresentam todas as suas juntas com a posição nula, esta função adapta os ângulos *position* para que correspondam correctamente aos servomotores da unidade *scuid* a que vão ser aplicados. Com base nas matrizes signal e offset – sinal de variação e desvio respectivamente, a posição final é calculada do seguinte modo: $position_{Final} = position \times signal + offset$ Verifique se estas matrizes possuem os valores correctos.
correct	`position_final=correct(scuid,position)` Correcção das posições *position* solicitadas para a unidade *scuid*, devido a desvios devido à imperfeita ligação dos servos à estrutura. Pequenos desvios da correia de transmissão ou do próprio veio do servomotor pode provocar deslocamentos, para uma determinada posição, ao longo do tempo. Actualize periodicamente a matriz *calib* de modo a reflectir esses desvios.
limit	`position_final=limit(scuid,position)` Limitação da posição *position* solicitada à unidade *scuid*. Além da limitação de excursão dos servos (-90 a +90º) a própria estrutura mecânica do robot aumenta ainda mais esta limitação. Confirme os dados da matriz *lim_min* e *lim_max*.

c) Visualização das Saídas dos Sensores de Força

Localização: **<CD_PROJ>:\Lab\Fase3_Integration\PC\Control\CoP_realtime**

Três rotinas de mais alto nível foram desenvolvidas tendo em vista o cálculo, armazenamento e representação gráfica do centro de pressão CoP (Tabela 27).

Tabela 27: Rotinas de alto-nível para medição das forças de reacção.

Funções M	***Descrição***
calccop	`CoP=calccop(react)` Com base nos dados sensoriais dos sensores de pressão (vector *react*) o centro de pressão *CoP* é calculado e retornado na forma de coordenadas rectangulares (*x*,*y*).
cop_realtime	(Rotina em formato script) Apresenta a localização do centro de pressão na base do pé, em tempo real, através de uma interface gráfica representativa de um pé.
cop_acquire	(Rotina em formato script) O armazenamento dos sensores de pressão (*reaction*) e do respectivo centro de pressão (*CoP*) é realizado durante um determinado período de tempo especificado pelo utilizador em *maxtime* (edit o *script*). Também a posição (*servo*) e a corrente consumida (*current*) da unidade slave local é capturada e armazenada. Entrada: *maxtime*: Tempo de captura Saídas: *reaction*: valores dos sensores de pressão (dimensão 4×TIME); *CoP*: localização (*x*,*y*) do centro de pressão (dimensão 2×TIME); *servo*: trajectória de posição realizada pelos 3 servos (dimensão 3×TIME) *current*: corrente consumida para os três servos (dimensão 3×TIME).
cop_graphs	(Rotina em formato script) Visualização gráfica de um ficheiro workspace com as variáveis produzidas pela rotina *cop_acquire*. Após produzir estas variáveis salvaguarde-as num ficheiro, e edite esta rotina para ler este ficheiro sempre que precisar.

Para testar novos algoritmos de controlo, sem ter que os implementar nas unidades de controlo, pode utilizar o simulador cinemático disponível em...

<CD_PROJ>:\Lab\Simulation\Simulator

... que imita o funcionamento de uma perna do robot humanóide. Todos os aspectos físicos presentes na perna, incluindo os dos servomotores, foram considerados para que a simulação seja o mais realista possível.

Edite e corra o script *simleg.m* para considerar as suas condições de funcionamento.

Índice de Figuras

Índice de tabelas