ELECTRONIQUE NUMERIQUE
MNE (M1)

Rapport : *Réalisation d'un système d'acquisition série pour PC*

Écrit par : Justin Silver

Groupe B (TP)

Introduction

Ce rapport décrit en détail le fonctionnement d'un système d'acquisition série pour un ordinateur hôte. Pour donner un résumé, le système à base d'un microcontrôleur PIC16F877 fonctionne de la manière suivante : Si le système est en mode automatique, toutes les secondes, un convertisseur analogique numérique échantillonne une tension, le microcontrôleur la convertit et la transcode en ASCII et ensuite l'envoie via liaison série (USART) à un ordinateur hôte. En mode manuel, une conversion et son envoi subséquent n'a lieu que si la demande est faite via une touche de clavier par l'utilisateur. Ces deux modes peuvent être choisis par l'utilisateur à n'importe quel moment après le démarrage de l'appareil.

Le descriptif simplifié du programme que le PIC tourne est décrit ci-dessous :

1) D'abord, une étape de configuration des périphériques (USART, Timer1, ADC) est exécutée
2) Ensuite, un message d'initialisation est envoyé via USART pour indiquer à l'utilisateur le bon démarrage du système
3) Le système entre dans une boucle infinie, en attendant la demande du mode de fonctionnement souhaité par l'utilisateur
4) Une fois la demande du mode est prise en compte, le système exécute les nombreuses étapes décrites dans le résumé du système
5) À partir de la troisième étape, tout se fonctionne en mode interruption (envoi/réception USART, overflow du Timer, conversion analogique numérique, etc.)

Les prochaines sections de ce rapport traiteront en plus de détail le programme et le fonctionnement du système d'acquisition.

Le rapport est divisé de la manière suivante :

- *Schéma bloc du système*
- *Code « relocatable »*
- *Périphériques*
- *Message d'initialisation*
- *Interrupt service routine (ISR)*
- *Problème : branchements conditionnels*
- *Callbacks*
- *Conversion en tension/ASCII*
- *Conclusion*

Schéma bloc du système

Ce schéma bloc fonctionnel décrit le flux du programme. Les différentes parties brièvement mentionnées dans l'introduction se retrouvent dans ce diagramme. Notamment, sur la branche gauche : une étape de configuration et ensuite un envoi d'un message de démarrage à l'utilisateur. Ensuite, le système entre dans une boucle infinie et en sort seulement s'il y a eu une interruption.

Si l'interruption RCIF a eu lieu, alors l'utilisateur a envoyé quelque chose via le terminal série (une demande de mode de fonctionnement). Il faut décider si le mode demandé est « automatique » ou « manuel » ou si l'utilisateur veut démarrer une conversion.

Si l'interruption TMR1F a eu lieu, le système est en mode automatique et il vérifie si une seconde s'est passée pour justifier ensuite le démarrage d'une conversion analogique numérique. Si ce n'est pas le cas, il faut attendre une autre interruption du Timer1.

Si l'interruption ADIF a eu lieu, la conversion analogique numérique est terminée et il faut convertir le résultat en tension et ensuite en ASCII pour l'envoi subséquent via USART.

Si l'interruption TXIF a eu lieu, il faut envoyer la prochaine byte des données ou terminer la transmission de données envers l'utilisateur via la liaison USART.

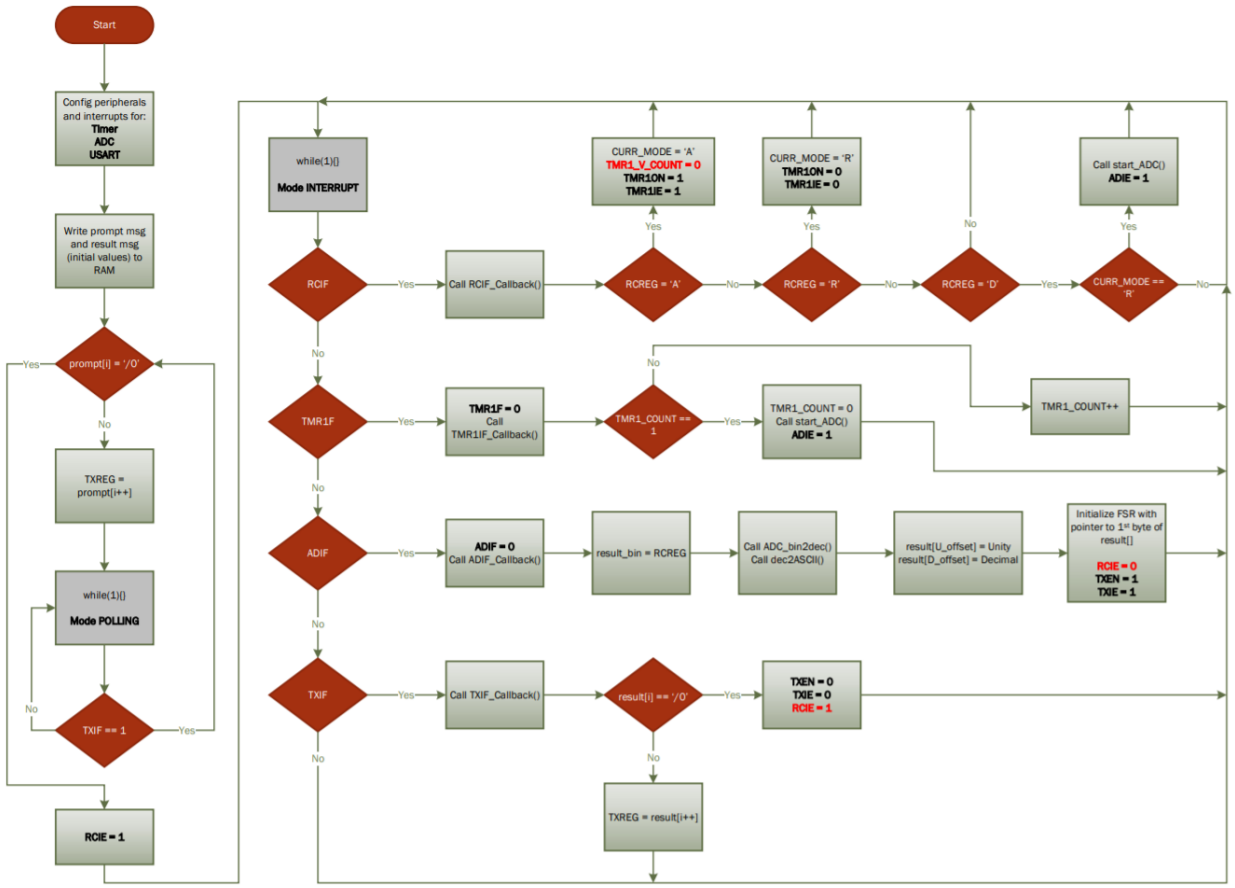

Figure 1. Schéma bloc du système et flux du programme

Code « relocatable »

Afin d'améliorer la lisibilité et pour simplifier la programmation/le débogage du système, plusieurs fichiers ou « modules » ont été construits pour chaque périphérique. La conséquence de cette décision est que le linker (pendant la compilation) décide où mettre les différentes sections du code en mémoire et où mettre les différentes variables (si ceci n'est pas explicitement précisées). Ainsi, les différents modules ne fonctionnent plus en code absolu, où l'adresse de chaque instruction est connue avant compilation.

Périphériques

Périphérique : ADC/CAN – convertisseur analogique numérique

Rôle : échantillonner une tension analogique qui a une valeur déterminée par la position d'un potentiomètre.

Configuration : Alignement sur ADRESH (8 bits de poids forts sont mis dans le registre ADRESH)

```
ADC_Config
        GLOBAL      ADC_Config
        banksel     ADCON1
        ; Left justify, 1 analog channel
        ; VDD and VSS references
        movlw       ( 0<<ADFM | 1<<PCFG3 | 1<<PCFG2 | 1<<PCFG1 | 0<<PCFG0 )
        movwf       ADCON1

        banksel     ADCON0
        ; Fosc/8, A/D enabled
        movlw       ( 0 << ADCS1 | 1<<ADCS0 | 1<<ADON )
        movwf       ADCON0
        return
```

Figure 2. Configuration du périphérique : convertisseur analogique numérique

---

Périphérique : USART

Rôle : Envoyer des données à l'ordinateur hôte et en recevoir les demandes par l'utilisateur (demande de mode automatique ou du mode manuel)

Configuration : Mode asynchrone, 9600 baud-rate. Pour configurer le générateur du baud-rate, il fallait d'abord passer par la configuration du prescaler en utilisant le calcul suivant :

```
; USART baud-rate prescaler value
; formula (asynchronous) = Baud-rate = FOSC/(16(X+1)) where X is the
; value in the SPBRG register and is between 0-255
; for a 9600 baud-rate with a FOSC of 4 MHz, we get X = 25
constant  X_VAL_SPBRG = D'25'
```

Figure 3. Configuration du prescaler du générateur du baud-rate de l'USART

```
USART_Config
    GLOBAL      USART_Config
    ; inputs/outputs (RC6/TX/CK and RC7/RX/DT)
    ; TXUSART is an output (RC6 -> output)
    ; RXUSART is an input (RC7 -> input)
    banksel     TRISC
    movlw       ( 0<<TRISC6 | 1<<TRISC7 )
    movwf       TRISC

    ; USART baud-rate setup
    banksel     SPBRG
    movlw       X_VAL_SPBRG
    movwf       SPBRG

    ; asynchronous mode
    ; HIGH SPEED mode (BRGH = 1) to reduce error on the baud-rate generation
    ; 8-bit transmission mode
    ; enable transmission (for polling prompt msg transmission)
    banksel     TXSTA
    movlw       ( 1<<TXEN | 0<<TX9 | 1<<BRGH | 0<<SYNC )
    movwf       TXSTA

    ; USART receiver setup
    banksel     RCSTA
    movlw       ( 1<<SPEN | 1<<CREN )
    movwf       RCSTA

    return
```

Figure 4. Configuration du périphérique : USART

Périphérique : Timer1

Rôle : Donner une cadence d'environ 500 ms grâce à une interruption d'overflow. Un registre à part compte si l'overflow a eu lieu 2 fois pour arriver à une cadence d'environ 1 seconde pour le mode automatique du système d'acquisition.

Configuration : La plus grande valeur du prescaler de ce compteur a été utilisé pour obtenir le maximum de temps entre des interruptions (~ 500 ms).

```
TMR1_Config
    GLOBAL      TMR1_Config
    banksel     T1CON
    ; Prescaler = 1:8
    ; Oscillator shut off
    ; Internal clock (Fosc/4) used

    ; Timer is NOT initially enabled (has to be enabled by USART receive MODE_REQUEST by user)
    ; as soon as the TMR1 module is enabled,
    ; it will start overflowing and the overflow flag will not be cleared. Thus, as soon
    ; as an interrupt flag comes, the TMR1IF will be checked and the TMR1IF_Callback
    ; will be immediately executed. This is not what we want -> We want TMR1 to only be counting
    ; once we're in automatic mode, otherwise it should be disabled
    movlw       ( 1<<T1CKPS1 | 1<<T1CKPS0 | 0 << TMR1CS | 0<<TMR1ON )
    movwf       T1CON
    clrf        TMR1_V_COUNT    ; the number of overflows is intialized to 0

    return
```

Figure 5. Configuration du périphérique : Timer1

Message d'initialisation

Avant d'envoyer un prompt à l'utilisateur, le string à envoyer est sauvegardé en mémoire. Les deux se font à l'aide de l'adressage indirect, en manipulant les registres : FSR et INDF. Le fonctionnement peut être résumé comme ceci :

- Tout d'abord, le registre FSR pointe au début du string qui sera envoyé plus tard dans le programme.
- Ensuite, le caractère ASCII souhaité à l'indice pointé par le FSR est chargé au registre de travail et subséquemment chargé au registre INDF. Cette dernière instruction active l'adressage en mode indirecte du microcontrôleur.
- L'indice du string pointé par le FSR est alors chargé avec ce qui a été mis dans le registre de travail et la valeur du registre du FSR est incrémenté pour pointer à la prochaine byte à envoyer
- Ces étapes répètent pour la longueur du string à envoyer et le dernier caractère sera = '/0' qui indique la fin du message

```
bankisel PTR_PROMPT_MSG
movlw    PTR_PROMPT_MSG
movwf    FSR

------------------------------------------------
        Initialize the prompt msg
------------------------------------------------

movlw A'C'
movwf INDF

incf FSR,F
movlw A'A'
movwf INDF

incf FSR,F
movlw A'N'
movwf INDF

incf FSR,F
movlw A' '
movwf INDF

incf FSR,F
movlw A'\r'
movwf INDF

incf FSR,F
movlw A'\n'
movwf INDF

incf FSR,F
movlw A'\0'
movwf INDF
```

Figure 6. Initialisation du message de démarrage en RAM

Le premier envoi d'un message à l'utilisateur se fait en mode polling et pas en mode interruption. C'est-à-dire que l'état du flag TXIF est polled pour savoir quand est-ce que la prochaine byte du string peut être envoyée.

La fin du message est déterminée par la rencontre de la caractère null ('\0'). Ceci évite la nécessité de compter le nombre de bytes envoyées et le comparer à la taille du string.

```
PRINT_PROMPT_MSG
    GLOBAL      PRINT_PROMPT_MSG
    ; a banksel AND a bankisel are necessary because the 2 different registers
    ; are accessed with 2 different methods (direct/indirect addressing)
    banksel     PIR1              ; set the appropiate values of (RP1, RP0)
    bankisel    PTR_PROMPT_MSG    ; set the appropiate value of (IRP)

    movlw       PTR_PROMPT_MSG
    movwf       FSR               ; point to the start of the prompt msg

TEST_END_OF_PROMPT_MSG
    movf        INDF, W           ; move current byte pointed by FSR to work reg
    xorlw       A'\0'             ; this operation will make Z flag = 1 if
                                  ; null character ('\0') of the msg is reached
    btfsc       STATUS, Z         ; if the end of the msg is reached, end USART comm
    goto        PROMPT_MSG_SENT

TEST_TXIF
    btfss       PIR1, TXIF        ; test if the TX_REG is empty
    goto        TEST_TXIF

    movf        INDF, W           ; place msg byte pointed to by FSR into work reg
    movwf       TXREG             ; send msg byte to USART TX register
    incf        FSR               ; increment pointer index to next byte in prompt msg
    goto        TEST_END_OF_PROMPT_MSG

; it is not necessary to preload the FSR and preset the IRP, a nop instruction could
; be placed here instead, or a rearranged labeling
PROMPT_MSG_SENT
                bankisel PTR_RESULT_MSG   ; preselect the correct bank for indirect addressing
                                          ; the start of this msg
                movlw PTR_RESULT_MSG      ; preload the FSR with the address to the
                movwf FSR                 ; ADC result (this will be the next msg to send)

                return
```

Figure 7. Envoi du message via USART en mode polling

Interrupt service routine (ISR)

Chaque fois qu'une interruption a lieu, après quelques instructions qui se font automatiquement par le hardware du processeur, le compteur du programme est mis à l'adresse 0x004. C'est à partir de cette adresse que les interruptions sont alors gérées.

Le contexte (état du registres STATUS, W, PCALTH) est sauvegardé temporairement avant de procéder.

```
ISR_FILE    CODE        0x004           ; interrupt vector location for the PIC uC

            movwf       W_TEMP          ; Copy W to TEMP register
            swapf       STATUS, W       ; Swap status to be saved into W
            movwf       STATUS_TEMP     ; Save status to bank zero STATUS_TEMP register
            movf        PCLATH, W
            movwf       PCLATH_TEMP
```

Figure 8. Sauvegarde du contexte dans l'ISR

Ensuite, comme illustré par le schéma bloc du système (Figure 1), l'origine de cette interruption est déterminée par des branchements conditionnels. Une fois l'interruption particulière est identifiée, un sous-programme « callback » est appelé pour le flag associé.

```
; Did we receive a message via USART?
RCIF_status
    banksel     PIR1
    btfss       PIR1, RCIF
    ; goto is necessary here because we are only skipping the NEXT
    ; instruction and not the entire long goto (lgoto)
    goto        TMR1IF_status       ; check next flag if this flag is not set
    lcall       RCIF_Callback
    PAGESEL     TMR1IF_status       ; in case previous subprogram call changed
                                    ; program memory page

; Did the Timer1 module overflow?
TMR1IF_status
    banksel     PIR1
    btfss       PIR1, TMR1IF        ; check next flag if this flag is not set
    goto        ADIF_status

    bcf         PIR1, TMR1IF        ; clear flag
    lcall       TMR1IF_Callback
    PAGESEL     ADIF_status         ; in case previous subprogram call changed
                                    ; program memory page

; Is the analog to digital conversion done?
ADIF_status
    banksel     PIR1
    btfss       PIR1, ADIF          ; check next flag if this flag is not set
    goto        TXIF_status

    bcf         PIR1, ADIF          ; clear flag
    lcall       ADIF_Callback
    PAGESEL     TXIF_status         ; in case previous subprogram call changed
                                    ; program memory page

; Is the USART TX register ready for a new character?
TXIF_status
    banksel     PIR1
    btfss       PIR1, TXIF
    goto        INT_FLAG_CHECK_DONE ; all flags checked

    lcall       TXIF_Callback
    PAGESEL     INT_FLAG_CHECK_DONE ; in case previous subprogram call changed
                                    ; program memory page
```

Figure 9. Arbre des branchements conditionnels dans l'ISR

Une remarque importante concerne l'utilisation des « long calls (lcall) » et des pseudo-instructions « PAGESEL ». Pour référence, « lcall » est tout simplement la pseudo-instruction PAGESEL combinée avec une instruction « call ».

Une conséquence de l'utilisation du code « relocatable » est que l'endroit en mémoire où chaque module existe n'est pas forcément connu avant la compilation du programme. Par conséquent, il faut obligatoirement mettre à jour le registre PCLATH avant d'appeler un sous-programme et/ou de faire un branchement « goto » pour pointer à la bonne page mémoire. Quand une de ces instructions est exécutée, une partie de l'adresse du PC est fournie par le registre PCLATH, et ce registre doit être configuré en software. La pseudo-instruction PAGESEL sélectionne la bonne page pour le label indiqué.

À la fin de l'ISR, le contexte est rechargé aux registres respectifs et l'instruction « return from interrupt enable » est exécutée.

```
INT_FLAG_CHECK_DONE
    swapf       STATUS_TEMP,W   ; Swap STATUS_TEMP register into W
                                ; (sets bank to original state)
    movwf       STATUS          ; Move W into STATUS register
    swapf       W_TEMP,F        ; Swap W_TEMP
    swapf       W_TEMP,W        ; Swap W_TEMP into W (restores original state)
    movf        PCLATH_TEMP, W  ; Restore PCLATH
    movwf       PCLATH
    retfie
```

Figure 10. Restauration du contexte dans l'ISR

Problème – branchements conditionnels

Une remarque très importante concerne la structure de ces conditionnels. Imaginons que le sous-programme destiné à démarrer le convertisseur analogique numérique est appelé. Une fois le convertisseur démarre, il faut attendre un certain nombre de cycles avant d'envoyer le caractère via USART. Avec cet arbre de branchements, il est alors nécessaire de désactiver l'USART pour éviter de rentrer dans le sous-programme destiné au flag TXIF.

Même si un des flags n'est pas activé à signaler une interruption au processeur, les flags sont toujours libres à changer leurs valeurs. Avec la structure choisie, l'état de chaque flag est testé, quel que soit l'origine de l'interruption et même après avoir appelé un callback/sous-programme pour un flag particulier.

La solution choisie pour résoudre ce problème a été de désactiver les périphériques et seulement les activer au moment approprié. Dans le cas du USART, si le transmetteur est désactivé, le TXIF sera = 0 et son callback ne sera pas appelé si ce n'est pas le bon moment dans le flux du programme.

Une autre solution pour contourner ce problème est proposé dans la dernière section de ce rapport : **Conclusion**.

Callbacks

RCIF

Ce callback est exécuté si le registre RCREG du périphérique USART reçoit une nouvelle byte du terminal série, ce qui indique au système que l'utilisateur choisit le mode du fonctionnement (automatique/manuel). Ce sous-programme permet de déterminer le mode souhaité.

Voici un exemple de comment le mode automatique est déterminé par l'envoi du caractère 'A' ou 'a' :

```
RCIF_Callback
    GLOBAL      RCIF_Callback
    banksel     RCREG
    movf        RCREG, W
    movwf       MODE_REQUEST        ; save what was received into a dedicated register
                                    ; this is to avoid possibly reading 2 different
                                    ; bytes in FIFO during the XOR tests below
TEST_IF_A_UPPER
    movf        MODE_REQUEST, W     ; move what was received into working reg
    xorlw       A'A'                ; this operation will make Z flag = 1 if
                                    ; the character 'A' was received
    btfsc       STATUS, Z
    goto        SET_AUTOMATIC_MODE

TEST_IF_A_LOWER
    movf        MODE_REQUEST, W     ; move what was received into working reg
    xorlw       A'a'                ; this operation will make Z flag = 1 if
                                    ; the character 'a' was received
    btfsc       STATUS, Z
    goto        SET_AUTOMATIC_MODE
```

Figure 11. Détection de la demande automatique par l'utilisateur

Une fois la demande est prise en compte, le mode automatique est activé et le périphérique Timer1 est mis en marche pour cadencer les envois des données.

```
SET_AUTOMATIC_MODE
    movlw       A'A'
    movwf       CURRENT_MODE

    ; the TMR1 and its interrupt are enabled at this point because in automatic mode
    ; the peripheral should be counting
    banksel     T1CON
    bsf         T1CON, TMR1ON
    banksel     PIE1
    bsf         PIE1, TMR1IE

    ; reset overflow count register (0 ms have passed)
    ; necessary in case RCIF is raised while TMR1_V_COUNT is at 1
    ; if that happens, TMR1_V_COUNT would never be reset
    banksel     TMR1_V_COUNT
    clrf        TMR1_V_COUNT
    goto        EXIT_RCIF_CALLBACK
```

Figure 12. Configuration du mode automatique

Le process pour gérer la demande du mode manuel et/ou pour la requête d'une conversion sont similaires.

TMR1F

Car le Timer1 ne peut générer une interruption qu'après 500 ms au maximum et une cadence de 1000 ms est nécessaire pour le mode automatique, il faut dédier un registre (TMR1_OVERFLOWED) pour compter le nombre de fois l'interruption a été générée. Par conséquent, le compteur du périphérique est essentiellement élargi.

```
TMR1IF_Callback
    GLOBAL      TMR1IF_Callback
    banksel     TMR1_V_COUNT

TEST_IF_SECOND_PASSED
    movf        TMR1_V_COUNT, W
    xorlw       TMR1_OVERFLOWED     ; this operation will make Z flag = 1 if
                                    ; the TMR1 module has already overflowed
                                    ; this would mean that approximately 1 second has passed
    btfsc       STATUS, Z
    goto SECOND_PASSED              ; this instruction is skipped if only 500 ms
                                    ; have occured
INCR_TMR1_V_COUNT
    incf        TMR1_V_COUNT, F     ; 500 ms have passed
    goto EXIT_TMR1IF_CALLBACK

SECOND_PASSED
    clrf        TMR1_V_COUNT            ; reset overflow count register (0 ms have passed)
    lcall       START_ADC
    PAGESEL     EXIT_TMR1IF_CALLBACK    ; not technically necessary to perform
                                        ; a PAGESEL psuedoinstruction here since no subsequent
                                        ; goto instructions are performed in this object module

EXIT_TMR1IF_CALLBACK
    return
```

Figure 13. Overflow du Timer1

TXIF

Similairement au mode polling, avant d'envoyer un nouveau caractère, la présence d'un caractère « null » est testé. Sa présence indique la fin du message.

```
TXIF_Callback
    GLOBAL      TXIF_Callback
    bankisel    PTR_RESULT_MSG    ; ensuring the (RP0, RP1) bits are correctly set/reset
                                  ; shouldn't really be necessary since
                                  ; indirect addressing should not occur
                                  ; elsewhere after printing the prompt message
TEST_END_OF_RESULT_MSG
    movf        INDF, W           ; move current byte pointed by FSR to work reg
    xorlw       A'\0'             ; this operation will make Z flag = 1 if
                                  ; null character ('\0') of the msg is reached
    btfsc       STATUS, Z         ; if the end of the msg is reached, end USART comm
    goto RESULT_MSG_SENT

SEND_NEW_BYTE
    banksel     TXREG
    movf        INDF, W           ; place msg byte pointed to by FSR into work reg
    movwf       TXREG             ; send msg byte to USART TX register
    incf        FSR               ; increment pointer index to next byte in result msg
    goto EXIT_TXIF_CALLBACK

RESULT_MSG_SENT
    banksel     TXSTA
    bcf         TXSTA, TXEN       ; disable USART transmission
    banksel     PIE1
    bcf         PIE1, TXIE
    bsf         PIE1, RCIE

EXIT_TXIF_CALLBACK
    return
```

Figure 14. Envoi des caractères en mode interruption

ADIF

Après qu'une conversion analogique numérique est finie (interruption par le flag ADIF), le résultat brut est sauvegardé en mémoire et ensuite converti en tension/caractère ASCII à l'aide des sous-programmes dans le même module (pour plus de détails sur ces conversions, consulter la section : **Conversion en tension/ASCII**.

```
ADIF_Callback
    GLOBAL      ADIF_Callback
    ; as long as there is no program memory boundary crossing in this module, no need to
    ; use a long call which selects the appropiate page bits
    call        ADC_BIN_TO_DEC_TO_ASCII
    ; once this function returns, the UNITY and DECIMAL values should be
    ; properly calculated and located in their dedicated registers
    ; we can now place these values into the appropiate indexes in the USART string
```

Figure 15. Appel aux sous-programmes pour la conversion du résultat ADC

Ensuite, les résultats ASCII sont placés dans le string à envoyer, et le pointeur pour ce string est initialisé à pointer au début du ce message.

```
; ---------------------------------
;  Place results in USART string
; ---------------------------------

    ; This section performs the following:
    ; result[U_offset] = Unity   (register)
    ; result[D_offset] = Decimal (register)
    bankisel PTR_RESULT_MSG   ; select the correct bank for indirect addressing
                              ; of the result message string
    movlw    (PTR_RESULT_MSG + UNITY_OFFSET)
    movwf    FSR              ; point to unity index in USART string
    movf     ADC_RESULT_UNITY, W
    movwf    INDF             ; place the ADC_UNITY_RESULT into the USART string
                              ; at the UNITY index
    movlw    (PTR_RESULT_MSG + DECIMAL_OFFSET)
    movwf    FSR              ; point to decimal index in USART string
    movf     ADC_RESULT_DECIMAL, W
    movwf    INDF             ; place the ADC_DECIMAL_RESULT into the USART string
                              ; at the DECIMAL index

; -----------------------------------------
;  Set FSR to point to start of USART string
; -----------------------------------------

    ; this is performed at the end, so that once the TXIF is raised, the FSR
    ; and IRP bits (bankisel) are good to go
    movlw PTR_RESULT_MSG      ; preload the FSR with the address to the
    movwf FSR                 ; ADC result (this will be the next msg to send)
```

Figure 16. Initialisation du string ASCII à envoyer

Finalement, la transmission et son interruption associée sont activées.

```
; a movlw then mowf operation
; cannot be done here, because the current mode of the system
; is not directly known. bit-wise operations are performed to avoid
; clobbering the state of the TMR1IE bit
banksel     PIE1
bcf         PIE1, RCIE   ; Receive USART flag disable
banksel     TXSTA
bsf         TXSTA, TXEN  ; transmission is now enabled
banksel     PIE1
bsf         PIE1, TXIE   ; USART TX interrupt flag enable

return
```

Figure 17. Configuration des périphériques après la conversion du résultat

Conversion en tension/ASCII

Le résultat de l'ADC est en binaire brut entre 0 et 255, ce qui correspond à une tension de 0 à 5 V. Le ratio pour le convertir en tension est le suivant :

$$tension = résultat \times \frac{5}{255} = résultat \times \frac{1}{51}$$

Cette division se fait à partir des instructions de soustraction et en regardant l'état du flag CARRY. Un exemple de comment les résultats de ces soustractions affectent l'état du CARRY est démontré ci-dessous :

```
; ------------------------------------------------------------------------
; Example/explanation: carry bit/NOT_borrow bit
; If we perform the subtract instruction -> 3-5, the ALU in the PIC uC performs
; a 2's complement on the 2nd operand and follows with an addition
; ------------------------------------------------------------------------
;   3  =   0011 =    3  =   0011   ->  3 in 2's complement and unsigned
; - 5  = - 0101 = +(-5) = + 1011   -> -5 in 2's complement, 11 in unsigned
;                       = 0 1110   =  -2 in 2's complement, 14 in unsigned
;               carry bit ^
; ------------------------------------------------------------------------
; What we see here is that if the result is supposed to be negative, the
; carry bit will be RESET. This will always be the case.
;
; However, if the inverse of this operation is done -> 5-3, we see that
; the result is expected to be positive and the 2's complement conversion
; is what leads to the carry bit being SET
; ------------------------------------------------------------------------
;   5  =   0101 =    5  =   0101  ->  5 in 2's complement and unsigned
; - 3  = - 0011 = +(-3) = + 1101  -> -3 in 2's complement, 13 in unsigned
;                       = 1 0010  =  -2 (in 2's complement), 18 in unsigned!
;               carry bit ^
; ------------------------------------------------------------------------
```

Figure 18. Explication et exemple de comment le flag CARRY fonctionne après une soustraction

Pour gagner en efficacité, la valeur initiale de l'UNITÉ et du DÉCIMAL = 0x30. Ceci car 0x30 en ASCII correspond au caractère '0', et chaque incrémentation d'1 augmente la valeur affichée en décimal. C'est-à-dire que 0x31 correspond au caractère '1', 0x32 = '2'… etc.

Après l'initialisation de l'UNITÉ et du DÉCIMAL, le résultat brut est sauvegardé en préparation des opérations de soustraction.

```
ADC_BIN_TO_DEC_TO_ASCII
    GLOBAL  ADC_BIN_TO_DEC_TO_ASCII

    movlw   (ASCII_NUMBER_OFFSET - 0x01)
    movwf   ADC_RESULT_UNITY            ; ADC_RESULT_UNITY currently contains 0x2F
    movwf   ADC_RESULT_DECIMAL          ; ADC_RESULT_DECIMAL currently contains 0x2F
                                        ; previous unity/decimal results are effectively erased
    ; pull result from A/D Result High Register (ADRESH)
    ; because the ADC is configured in left justified,
    ; we should be pulling the 8 MSBs from the 10-bit result
    banksel ADRESH
    movf    ADRESH,W
    movwf   ADC_RESULT_BINARY           ; save to dedicated register for subsequent subwf instructions
```

Figure 19. Initialisation et préparation de la conversion binaire à ASCII

Ensuite, la conversion est effectuée pour l'UNITÉ et pour le DÉCIMAL.

```
; ------------------------------------
;           UNITY calculation
; ------------------------------------

    ; first pass through this instruction block, ADC_RESULT_UNITY
    ; will become 0x30 (0 in ASCII)
    ; by preloading this register with ASCII_NUMBER_OFFSET - 1, the initial value is
    ; effectively "set" to 0 in ASCII, without the need for extra gotos
CALCULATE_UNITY_PLACE
    incf    ADC_RESULT_UNITY, F
    movwf   ADC_RESULT_BINARY           ; save result of substraction (division "remainder")
    movlw   ADC_BIN_VOLT_UNITY_RATIO    ; prepare substraction constant value in working reg

    subwf   ADC_RESULT_BINARY, W        ; subtract 51 from binary result (1 volt corresponds to 51 in dec (ratio))
    btfsc   STATUS, C                   ; if the carry bit is SET, the result is POSITIVE
    goto    CALCULATE_UNITY_PLACE       ; this means that the UNITY value for the ADC result is not yet found
```

Figure 20. Calcul de la valeur de l'UNITÉ

```
CALCULATE_DECIMAL_PLACE
    incf    ADC_RESULT_DECIMAL, F
    movwf   ADC_RESULT_BINARY           ; save result of substraction (division "remainder")
    movlw   ADC_BIN_VOLT_DECIMAL_RATIO  ; prepare substraction constant value in working reg

    subwf   ADC_RESULT_BINARY, W        ; subtract 5 from binary result (0.1 volt corresponds to 5 in dec (ratio))
    btfsc   STATUS, C                   ; if the carry bit is SET, the result is POSITIVE
    goto    CALCULATE_DECIMAL_PLACE     ; this means that the DECIMAL value for the ADC result is not yet found
```

Figure 21. Calcul de la valeur du DÉCIMAL

Un peu de résolution est perdue avec le calcul de la décimale, car au lieu de diviser le reste par 5,1 pour obtenir la partie décimale de la tension, le reste est « divisé » par 5.

Une conséquence de ceci est le cas spécial quand le reste = 50. Ce cas est spécial car si le même algorithme est utilisé pour le calcul de la décimale, la décimale va incrémenter 10 fois. Or, dès le départ de l'algorithme, la valeur de la décimale intègre déjà l'offset ASCII. Donc, si ce cas n'est pas traité, le résultat final de la décimale va = « : » en ASCII, et il sera affiché ainsi à l'utilisateur.

La solution adaptée pour ce système est de simplement arrondir à la prochaine valeur. Autrement dit, 0.9 devient 1.0, 1.9 devient 2.0, et ainsi de suite.

```
REMAINDER_IS_50
    movlw   A'0'
    movwf   ADC_RESULT_DECIMAL          ; the decimal place automatically takes the ASCII character 0
    incf    ADC_RESULT_UNITY, F         ; round up the result (the unity is incremented)
                                        ; this means that 4.9 would become 5, 3.9 would become 4, etc...
    goto    EXIT_CONVERSION
```

Conclusion

Le rapport intègre des explications des différents modules pour ce système et comment ils fonctionnent ensemble.

Pour la suite de ce projet, il y a plusieurs améliorations qui pourraient être explorées.

1) Pendant que le processeur ne fait rien dans le programme principal, (boucle infinie) il serait prudent à le mettre en mode sleep/low-power pour conserver de l'énergie.

2) Une autre amélioration pourrait être de l'enlever le mode polling de l'envoi du prompt et de mettre cela directement dans le sous-programme TXIF callback. Il faudrait alors nécessaire à distinguer entre le prompt à envoyer et le résultat de tension.

3) Si plus de résolution sur le niveau de tension est souhaitée, le programme pourrait être modifié d'incorporer le registre ADRESL qui contient les 2 bits de poids faible non-utilisés dans ce système.

4) Concernant le problème des branchements conditionnels dans l'ISR, une solution différente serait d'ajouter un bit test pour le bit d'activation du flag d'interruption du périphérique visé (par exemple le bit TXIE). Si ce bit = 1, alors le flag d'interruption sera testé. Sinon, la prochaine source d'interruption éventuelle sera testée. Ensuite, après l'appel d'un callback, un « goto » est utilisé pour aller à la fin de l'ISR et pour quitter l'arbre des branchements. Avec cette solution, un seul callback peut être appelé et ce ne sera le cas que si l'interruption pour le flag est activée et l'interruption elle-même a eu lieu. Ceci évite de tester la source d'une interruption qui ne devrait pas être testée.

5) Finalement, plus d'indications visuelles pourraient être implémentées pour l'utilisateur. Par exemple, chaque fois qu'un nouveau mode est sélectionné, un message peut être envoyé via USART qui indique sur le terminal le mode actuel (automatique ou à la demande). Dans la même idée, si l'utilisateur choisit le mode manuel, un message de prompt qui lui indique à taper la commande r ou R pourrait être envoyé.