The 555.m4x and RSB.m4x SynDEx Macro-Executives: Description of Macros for Handling RSMPC555 Boards

Pierre Pomiers
Robosoft S.A.
Technopole d'Izarbel, 64210 Bidart, France.
Tel.: +33-5-59 41 53 66; Fax.: +33-5-59 41 53 79
E-mail: pierre@robosoft.fr

September 14, 2005

Contents

List of Figures

List of Tables

Part I

Acknowledgments

Introduction

This document applies to programmers intended to use SynDEx [1] for programming real-time architectures based on RSMPC555 Robosoft control boards. Relying on the AAA[1] methodology, SynDEx is known to be both a gentle tool for distributed architectures handling and a good way of optimizing hardware resources management. In order to take benefits of these capabilities for our RSMPC555 product, we developped a specific SynDEx macro-executive. It gathers all macros you need for handling Motorola MPC555 chip functionnalities and more.

You will find here a clear documentation of all these macros: definition, domain of use for there respective input and (or) output ports), as well as there basic principle of execution. Moreover, for advanced users, we give some useful related links toward data-sheets, manufacturers web sites and online documentations.

We intentionally have split this document into two main parts, one for each of the 555.m4x and RSB.m4x macro-executives. The first chapter focuses Motorola MPC555 bare chip related macros, while the second one refer to macros specifically designed for our own board hardware.

Let us remark that, from chapter 6 to document end, described macros refer to the MPC555 input/output features. As far as this part of the document is concerned, we classify macros into two different cathegories: "generic" macros and "user" macros. By "generic" we mean macros that are not intended to be used directly from the SynDEx CAD interface, but macros used for programming other convenient macros. Indeed, most of the time, they give control over many parameters non-expert users are not intended to manipulate. In other words, "generic" macros should only be used by persons aiming to program new user-oriented macros. On the other hand, by "user" macros we mean macros that may be used directly from the SynDEx CAD interface. These macros are seen as user-oriented operations. They embed their own context switch, what is not the case with "generic" macros. "user" macros context switch, used in conjunction with the SynDEx macro generation context variable `MGC`, aims to provide the appropriate part of executive, depending on the execution step macro is currently called from. It exists three contexts: application initialization (with `MGC` equal to `INIT`), loop execution (with `MGC` equal to `LOOP`) and application finalization (with `MGC` equal to `END`).

We highly recommand users to refer to SynDEx web site ([1]). You will find here all information needed for programming with SynDEx, a SynDEx CAD interface tutorial, as well as many related links.

[1] Algorithm Architecture Adequation

Recommandations

The remainder of this document lists and describes the SynDEx MPC555 and Robosoft board macro set. Before getting to the following pages, please refer to figure 1 that show the basic structure of macros descriptions.

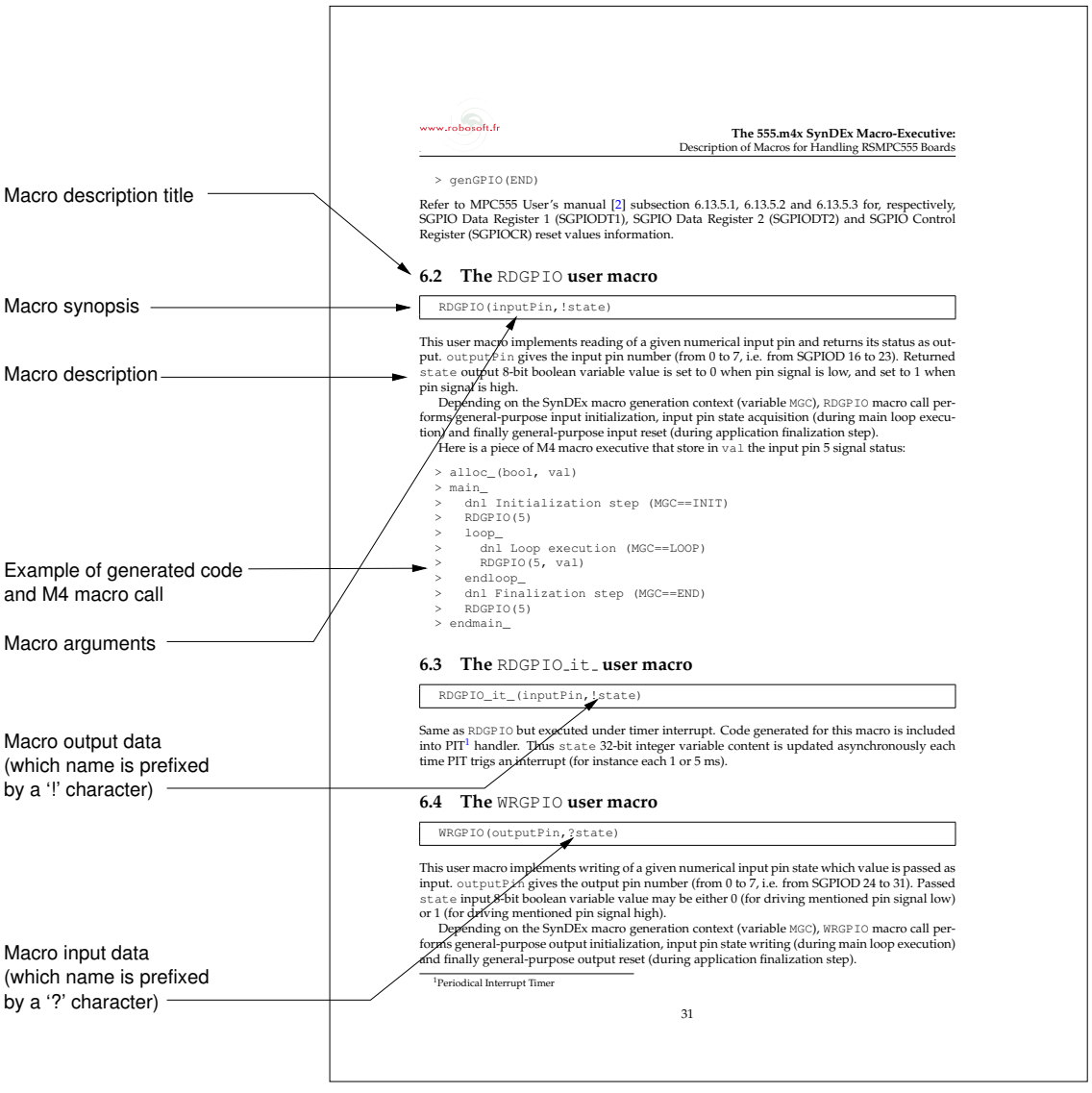

Figure 1: Structure of macro description

Part II

Motorola MPC555 chip specific macros

Chapter 1

Subroutine calls for interfacing separately compiled C functions

For the MPC555, when calling a separately compiled C functions, first function parameters are passed directly in the registers and the remaining parameters are passed in the stack.

When using GNU PowerPC ELF 32-bit cross compiler, integer registers `r3`, `r4`, ..., `r10` are allocated in this order for unsigned or signed integers (char, short, int) and for addresses. The same way, floating point registers `fr1`, `fr2`, ..., `fr8` are allocated in this order for single and double precision reals (float, double). In the case the function returns a result, it is passed back in register `r3` if integer, or in register `fr1` if real.

Be careful, in the present implementation, only register passing is supported. A message error indicating too many arguments are passed is generated if registers allocation is over.

1.1 The `Cdecl_` macro

```
Cdecl_($1, $2, $3, ..., $N)
```

This macro generates the declaration of a separately C compiled function. When calling `Cdecl_`, `$1` is used for passing return value type (int, float, double, etc...), `$2` is used for passing C function name and `$3` up to `$N` for passing function arguments type (postfixed by '`*`' character if it indicates an argument passed by address). Here is an example of M4 macro-call:

```
> Cdecl_(int,my_fun,int,int*,float)
```

and the corresponding generated MPC555 compatible assembly code:

```
.global my_fun # int my_fun(int, int*, float);
```

This macro is useful when your compiler needs to be informed of a exixting separately C compiled function for completing compilation process.

1.2 The `Ccall_` macro

```
Ccall_($1, $2, $3, ..., $N)
```

This macro mimics the ANSI C declaration of a separately compiled function. When calling `Ccall_`, `$1` is used for passing return value type and label (or `void` if none) and `$2` is used for passing C function name. `$3` up to `$N` correspond to either `const` followed by a literal for an integer literal argument, or argument type (postfixed by '`*`' character if passed by address) followed by a label. Here is an example of M4 declaration and macro-call:

```
> def('myfun', 'Ccall_(int $3, my_fun, const 5, int *$1, float $2)')
> myfun(arg1,arg2,res)
```

and the corresponding generated MPC555 compatible assembly code:

```
li r3,5
B(la r4,arg1)
B(lfs fr1,arg2)
bl my_fun
B(stw r3,res)
```

Chapter 2

Generic unary operations for standard scalar and array types

2.1 The `gnot` macro

```
gnot(?X, !res)
```

This macro implements the bitwise negation. It sets `res` with the one's complement of `X`.

2.2 The `gneg` macro

```
gneg(?X, !res)
```

This macro implements the negation. It sets `res` with the negative of `X`.

Chapter 3

Generic binary operations for standard scalar and array types

3.1 The `gand` macro

```
gand(?x, ?y, !res)
```

This macro implements the bitwise AND operation. It sets `res` with the result of `x` ANDed with `y`.

3.2 The `gor` macro

```
gor(?x, ?y, !res)
```

This macro implements the bitwise OR operation. It sets `res` with the result of `x` ORed with `y`.

3.3 The `gxor` macro

```
gxor(?x, ?y, !res)
```

This macro implements the bitwise exclusive OR operation. It sets `res` with the result of `x` XORed with `y`.

3.4 The `gadd` macro

```
gadd(?x, ?y, !res)
```

This macro implements the addition operation. It sets `res` with the result of `x` added to `y`.

3.5 The `gsub` macro

```
gsub(?x, ?y, !res)
```

This macro implements the substraction operation. It sets `res` with the result of `y` substracted from `x`.

3.6 The `gmul` macro

```
gmul(?x, ?y, !res)
```

This macro implements the multiplication operation. It sets `res` with the result of `x` multiplied by `y`.

3.7 The `gdiv` macro

```
gdiv(?x, ?y, !res)
```

This macro implements the division operation. It sets `res` with the result of `x` divided by `y`.

Chapter 4

Generic relational operations for standard scalar types

4.1 The `gequal` macro

```
gequal(?x, ?y, !res)
```

This macro implements the equality operation. It sets `res` to -1 in case of equality between `x` and `y`, else, sets `res` to 0.

4.2 The `gnotequal` macro

```
gnotequal(?x, ?y, !res)
```

This macro implements the non-equality operation. It sets `res` to -1 in case of non-equality between `x` and `y`, else, sets `res` to 0.

4.3 The `gless` macro

```
gless(?x, ?y, !res)
```

This macro implements the "strictly less than" operation. It sets `res` to -1 if `x` value is less than `y` value, else, sets `res` to 0.

4.4 The `gnotless` macro

```
gnotless(?x, ?y, !res)
```

This macro implements the "greater than or equal to" operation. It sets `res` to -1 if `x` value is greater than or equal to `y` value, else, sets `res` to 0.

Chapter 5

Generic DSP-like operations for integer and float types

5.1 The `gdotProd` macro

```
gdotProd(?X, ?Y, !res)
```

This macro implement the dot product of `X` and `Y`. `X` and `Y` type may be either int, float, double, array of int, array of float or array of double. Result of the dot product is returned in `res` of relevant type.

$$\texttt{res} = \sum_{i=0}^{N} \texttt{X}[i] * \texttt{Y}[i]$$

Note that when used for FIR or IIR filters, one of `X` or `Y` is a slidding window, the other is the coefficients array. For IIR filters, the input and output slidding windows are concatenated as are the two coefficient arrays, and the result is stored at the array end.

5.2 The `gequalize` macro

```
gequalize(?err, ?win, ?!coef)
```

This macro implements an equalizer algorithm, where `err` is used for passing an error value and `win` for passing a window. `coeff` is both an input and output argument as it is used for updating a coefficient or an array of coefficient. Result of the equalization is returned in `coeff`. In the following equation, n indicates the current application cycle number.

$$\begin{bmatrix} \texttt{coeff}_n[0] \\ \vdots \\ \texttt{coeff}_n[i] \\ \vdots \\ \texttt{coeff}_n[N-1] \end{bmatrix} = \begin{bmatrix} \texttt{coeff}_{n-1}[0] \\ \vdots \\ \texttt{coeff}_{n-1}[i] \\ \vdots \\ \texttt{coeff}_{n-1}[N-1] \end{bmatrix} - \left(\begin{bmatrix} \texttt{win}[0] \\ \vdots \\ \texttt{win}[i] \\ \vdots \\ \texttt{win}[N-1] \end{bmatrix} * \texttt{err} \right)$$

Note that `err` type may be either int, float or double, while `win` and `coeff` type may be array of int, array of float or array of double.

Chapter 6

General-purpose input/output

The MPC555 incorporates system functions that normally must be provided in external circuits. For, through the USIU submodule, it provides 64 pins for general-purpose digital I/O. The SGPIO pins are multiplexed with the address and data pins. As, in our case, SGPIO pins are not required for communicating with external devices is not required, the user can freely use all of them.

6.1 The `genGPIO` generic macro

6.1.1 The `INIT` option

```
genGPIO(INIT)
```

Called with `$1` equal to `INIT`, this macro initializes the MPC555 General-purpose input/output ports. After initialization, SGPIOD pins 16 to 23 are set as inputs and SGPIOD pins 24 to 31 are set as outputs.

Here is the M4 macro-call used for initialization:

```
> genGPIO(INIT)
```

Refer to MPC555 User's manual [2] section 6.3 for hardware details.

6.1.2 The `RDSTATE` option

```
genGPIO(RDSTATE, ?int)
```

Called with `$1` equal to `RDSTATE`, this macro read, in one shot, the 8 input pins status (from SGPIOD pins 16 to 23) and reports it in the passed `int` 8-bit boolean variable. If returned value bit k (from 0 to 7) is set to 0 then SGPIOD pin 16+k signal is low, else, if set to 1, SGPIOD pin 16+k signal is high.

Here is an example of M4 macro-call that store the 8 input pins status into the passed variable `val`:

```
> genGPIO(RDSTATE, val)
```

Refer to MPC555 User's manual [2] subsection 6.13.5 for further information.

6.1.3 The WRSTATE option

```
genGPIO(WRSTATE, ?int)
```

Called with $1 equal to WRSTATE, this macro write, in one shot, the 8 output pins states (from SGPIOD pins 24 to 31) according to value passed in int 8-bit boolean variable. If passed value bit *k* (from 0 to 7) set to 0 SGPIOD pin 24+*k* signal is driven low, else, if set to 1, SGPIOD pin 24+*k* signal is driven high.

Here is an example of M4 macro-call that drives output pin 1 and output pin 2 signals high, other pins signals are driven low:

```
> genGPIO(WRSTATE, val) # With val set to 6 (00000110b).
```

Refer to MPC555 User's manual [2] subsection 6.13.5 for further information.

6.1.4 The RDPIN option

```
genGPIO(RDPIN, pin, !bool)
```

Called with $1 equal to RDPIN, this macro read the status of input pin which number is given in pin argument. pin argument value may be from 0 to 7 corresponding to SGPIOD pins from 16 to 23. Status value is reported into the passed bool 8-bit boolean variable. If mentioned input pin signal is low then bool is set to 0, else, if mentioned input pin signal is high, bool is set to 1.

Here is an example of M4 macro-call that store input pin 3 status into the passed variable val:

```
> genGPIO(RDPIN, 3, val) # With val set to 6 (00000110b).
```

Refer to MPC555 User's manual [2] subsection 6.13.5 for further information.

6.1.5 The WRPIN option

```
genGPIO(WRPIN, pin, ?bool)
```

Called with $1 equal to WRPIN, this macro write one pin state. Pin number is given in pin argument that may be from 0 to 7 corresponding to SGPIOD pins from 24 to 31. State value is passed in the 8-bit boolean variable bool. If passed value is set to 0 then mentioned output pin is driven low, else, if passed value is set to 1 then mentioned output pin is driven high. Other not mentionned pins signals are left unchanged.

Here is an example of M4 macro-calls that drives output pin 4 signal low, output pin 7 signal high and leaves other pins signals unchanged: passed variable val:

```
> genGPIO(WRPIN, 4, val0) # With val0 set to 0.
> genGPIO(WRPIN, 7, val1) # With val1 set to 1.
```

Refer to MPC555 User's manual [2] subsection 6.13.5 for further information.

6.1.6 The END option

```
genGPIO(END)
```

Called with $1 equal to END, this macro reset SIU general-purpose I/O groups directions and data to their initial states.

Here is the M4 macro-call used for finalization:

```
> genGPIO(END)
```

Refer to MPC555 User's manual [2] subsection 6.13.5.1, 6.13.5.2 and 6.13.5.3 for, respectively, SGPIO Data Register 1 (SGPIODT1), SGPIO Data Register 2 (SGPIODT2) and SGPIO Control Register (SGPIOCR) reset values information.

6.2 The RDGPIO user macro

```
RDGPIO(inputPin,!state)
```

This user macro implements reading of a given numerical input pin and returns its status as output. `outputPin` gives the input pin number (from 0 to 7, i.e. from SGPIOD 16 to 23). Returned `state` output 8-bit boolean variable value is set to 0 when pin signal is low, and set to 1 when pin signal is high.

Depending on the SynDEx macro generation context (variable `MGC`), `RDGPIO` macro call performs general-purpose input initialization, input pin state acquisition (during main loop execution) and finally general-purpose input reset (during application finalization step).

Here is a piece of M4 macro executive that store in `val` the input pin 5 signal status:

```
> alloc_(bool, val)
> main_
>   dnl Initialization step (MGC==INIT)
>   RDGPIO(5)
>   loop_
>     dnl Loop execution (MGC==LOOP)
>     RDGPIO(5, val)
>   endloop_
>   dnl Finalization step (MGC==END)
>   RDGPIO(5)
> endmain_
```

6.3 The RDGPIO_it_ user macro

```
RDGPIO_it_(inputPin,!state)
```

Same as `RDGPIO` but executed under timer interrupt. Code generated for this macro is included into PIT[1] handler. Thus `state` 32-bit integer variable content is updated asynchronously each time PIT trigs an interrupt (for instance each 1 or 5 ms).

6.4 The WRGPIO user macro

```
WRGPIO(outputPin,?state)
```

This user macro implements writing of a given numerical input pin state which value is passed as input. `outputPin` gives the output pin number (from 0 to 7, i.e. from SGPIOD 24 to 31). Passed `state` input 8-bit boolean variable value may be either 0 (for driving mentioned pin signal low) or 1 (for driving mentioned pin signal high).

Depending on the SynDEx macro generation context (variable `MGC`), `WRGPIO` macro call performs general-purpose output initialization, input pin state writing (during main loop execution) and finally general-purpose output reset (during application finalization step).

[1] Periodical Interrupt Timer

Here is a piece of M4 macro executive that drives output pin 5 signal, depending on input variable `val`:

```
> alloc_(bool, val)
> main_
>   dnl Initialization step (MGC==INIT)
>   WRGPIO(5)
>   loop_
>     dnl Loop execution (MGC==LOOP)
>     WRGPIO(5, val)
>   endloop_
>   dnl Finalization step (MGC==END)
>   WRGPIO(5)
> endmain_
```

6.5 The `WRGPIO_it_` user macro

```
WRGPIO_it_(outputPin,?state)
```

Same as `WRGPIO` but executed under timer interrupt. Code generated for this macro is included into PIT handler. Thus `state` 32-bit integer variable content is read asynchronously each time PIT trigs an interrupt.

Chapter 7

Queued analog to digital converter module

Analog to digital conversion is an electronic process in which a continuously variable analog signal is changed, without altering its essential content, into a multi-level digital signal. The input to an analog to digital converter (ADC) consists of a voltage that varies among a theoretically infinite number of values. The output of the ADC, in contrast, is defined over 2 states as a binary digital signal.

7.1 The `genQADC` generic macro

7.1.1 The `INIT` option

```
genQADC(INIT, analogChannel, inputSampleTime)
```

Called with `$1` equal to `INIT`, this macro initializes the MPC555 QADC (Queued Analog-to-Digital Converter) Module A and B. Accessible analog channels are numbered from 0 to 31. Correspondance between analog channels numbers and MPC555 hardware pins is given in table 7.1 (see also MPC555 User's manual [2] Table 13-20): analog channels numbers are passed using argument `analogChannel`.

User can also specifies the window length to be used for analog channel sampling. Longer sample times permit more accurate A/D conversions of signals with higher source impedances (refer to MPC555 User's manual [2] Table 13-19). This choice is done setting `inputSampleTime` argument. Possible value are 2, 4, 8 or 16, corresponding to the following sample window length: $(T_{QCKLs} * 2)$, $(T_{QCKL} * 4)$, $(T_{QCKL} * 8)$ or $(T_{QCKL} * 16)$, where T_{QCKL} is the QADC64 Clock period. Calculation of T_{QCKL} is explained in MPC555 User's manual [2] subsection 13.10.4 at page 13-27). In current implementation, QCLK frequency is set to 2 MHz (i.e. $T_{QCKL} = 500\ ns$).

Analog channel number	Reference to MPC555 hardware
0 to 3	QADC Module A channels 0 to 3
4 to 15	QADC Module A channels 48 to 59
16 to 19	QADC Module B channels 0 to 3
20 to 31	QADC Module B channels 48 to 59

Table 7.1: ADC Channel number assignments and pin designations

Here is the M4 macro-call used for initializing analog channel 4 and selecting a $(T_{QCKL} * 8)$ sample window length:

```
> genQADC(INIT, 4, 8)
```

Refer to MPC555 User's manual [2] subsection 13.12 for programming model details.

7.1.2 The LOOP option

```
genQADC(LOOP, analogChannel, inputSampleTime, !int)
```

Called with `$1` equal to `LOOP`, this macro reads the conversion result obtained from sampling analog channel signal. Conversion result is coded over 10 bits. Analog channel number is passed using argument `analogChannel`. Conversion result is aligned and stored into the passed 32-bit interger variable `int`. Let us remark that Robosoft board hardware fix the analog input low voltage reference to $V_{RL} = 0V$ and high voltage reference to $V_{RH} = 5V$. Corresponding returned conversion values range are between 0 and 1023.

Here is an example of M4 macro-call used for reading analog channel 5 signal converted value:

```
> genQADC(LOOP, 2, 5, val)
```

Refer to MPC555 User's manual [2] subsection 13.12.11 for details concerning command conversion.

7.1.3 The END option

```
genQADC(END, analogChannel, inputSampleTime)
```

Called with `$1` equal to `END`, this macro reset analog channel which number is passed using argument `analogChannel`. It consistes of finishing any current conversion and then freezing QADC module (refer to [2] subsection 13.12.1).

Here is an example of M4 macro-call used for analog channel 1 finalization:

```
> genQADC(END, 1)
```

7.2 The RDQADC user macro

```
RDQADC(analogChannel,inputSampleTime,!result)
```

This user macro implements reading of a given analog channel signal conversion result. Input channel number (from 0 to 31) is passed using argument `analogChannel`. Conversion result (coded over 10 bits) is aligned and stored into the passed 32-bit interger variable `result`. Refer to previous section for details about `inputSampleTime` settings.

Depending on the SynDEx macro generation context (variable `MGC`), `RDQADC` macro call performs QADC module initialization, analog channel signal conversion (during main loop execution) and finally QADC module reset (during application finalization step).

Here is a piece of M4 macro executive that reads the conversion result of analog channel 1 and returns it into variable `val`:

```
> alloc_(int, val)
> main_
>   dnl Initialization step (MGC==INIT)
>   RDQADC(1, 2)
>   loop_
>     dnl Loop execution (MGC==LOOP)
```

```
>     RDQADC(1, 2, val)
>   endloop_
>   dnl Finalization step (MGC==END)
>   RDQADC(1, 2)
> endmain_
```

7.3 The RDQADC_it_ user macro

```
RDQADC_it_(analogChannel,inputSampleTime,!result)
```

Same as RDQADC but executed under timer interrupt. Code generated for this macro is included into PIT handler. Thus `result` 32-bit integer variable content is updated asynchronously each time PIT trigs an interrupt.

Chapter 8

MIOS 16-bit Parallel Port I/O Submodule

The MPC555 incorporates system functions that normally must be provided in external circuits. For, through the MIOS submodule (MPIOSM), it provides 16 pins for digital I/O. On the Robosoft board, all of the MPIOSM pins are set as output (i.e. for sending digital signal). As in our case, some pins are normally required for driving external devices user can not freely use all of them.

8.1 The `genMPIOSM` generic macro

8.1.1 The `INIT` option

```
genMPIOSM(INIT)
```

Called with `$1` equal to `INIT`, this macro initializes the MPC555 MPIOSM 16-bits port. After initialization, SGPIOD pins 0 to 15 are set as output.

Here is the M4 macro-call used for initialization:

```
> genMPIOSM(INIT)
```

Refer to MPC555 User's manual [2] section 15.13 for hardware details.

8.1.2 The `LOOP` option

```
genMPIOSM(LOOP,pin,?bool)
```

Called with `$1` equal to `LOOP`, this macro initializes the MPC555 MPIOSM 16-bits port. After initialization, SGPIOD pins 0 to 15 are set as output.

Called with `$1` equal to `LOOP`, this macro write one pin state. Pin number is given in `pin` argument that may be from 0 to 15 corresponding to MPIOSM pins from 0 to 15. State value is passed in the 8-bit boolean variable `bool`. If passed value is set to 0 then mentioned output pin is driven low, else, if passed value is set to 1 then mentioned output pin is driven high. Other not mentionned pins signals are left unchanged.

Here is an example of M4 macro-calls that drives output pin 4 signal low, output pin 7 signal high and leaves other pins signals unchanged: passed variable `val`:

```
> genMPIOSM(LOOP, 4, val0) # With val0 set to 0.
> genMPIOSM(LOOP, 7, val1) # With val1 set to 1.
```

Refer to MPC555 User's manual [2] subsection 15.13.1 for further information.

8.1.3 The `ON` option

```
genMPIOSM(ON,pin)
```

Called with `$1` equal to `ON`, this macro drive a given MPIOSM port pin state high. Pin number is passed using argument `pin` which possible values are from 0 to 15.

Here is an example of M4 macro-calls that drives output pin 4 signal high and leaves other pins signals unchanged: passed variable `val`:

```
> genMPIOSM(ON, 4)
```

8.1.4 The `OFF` option

```
genMPIOSM(OFF,pin)
```

Called with `$1` equal to `OFF`, this macro drive a given MPIOSM port pin state low. Pin number is passed using argument `pin` which possible values are from 0 to 15.

Here is an example of M4 macro-calls that drives output pin 7 signal low and leaves other pins signals unchanged: passed variable `val`:

```
> genMPIOSM(OFF, 7)
```

8.1.5 The `END` option

```
genMPIOSM(END)
```

Called with `$1` equal to `END`, this macro resets the MPC555 MPIOSM 16-bits port. After finalizations, MPIOSM pins 0 to 15 are set as input.

Here is the M4 macro-call used for initialization:

```
> genMPIOSM(INIT)
```

Refer to MPC555 User's manual [2] section 15.13 for hardware details.

8.2 The `MPIOSM` user macro

```
WRGPIO(outputPin,?state)
```

This user macro implements writing of a given numerical state which value is passed as input. `outputPin` gives the output pin number (from 0 to 15, i.e. from MPIOSM 0 to 15). Passed `state` 8-bit boolean input variable value may be either 0 (for driving mentioned pin signal low) or 1 (for driving mentioned pin signal high).

Depending on the SynDEx macro generation context (variable `MGC`), `MPIOSM` macro call performs general-purpose output initialization, input pin state writing (during main loop execution) and finally general-purpose output reset (during application finalization step).

Here is a piece of M4 macro executive that drives output pin 5 signal, depending on input variable `val`:

```
> alloc_(bool, val)
> main_
>   dnl Initialization step (MGC==INIT)
>   MPIOSM(5)
>   loop_
```

```
>     dnl Loop execution (MGC==LOOP)
>     MPIOSM(5, val)
>   endloop_
>   dnl Finalization step (MGC==END)
>   MPIOSM(5)
> endmain_
```

8.3 Note for users

Let us note that MPIOSM pins from 0 to 11 may be used bye other macros. Table 8.1 give information about possible dependencies. Anyway, if your application contains no call to the macros mentionned in this table, you can make use of MPIOSM pins freely.

MPIOSM pins	Attached signal	Macros
0	SPI encoder reference reset	`SPIencoder_it_`, `SPIencoder` and `genSPIencoder`
1	SPI encoder LDR signal	`DAC_it_`, `DAC` and `genDAC`
2	SPI encoder SCK signal	`DAC_it_`, `DAC` and `genDAC`
3	SPI encoder SDI signal	`DAC_it_`, `DAC` and `genDAC`
From 4 to 7	Amp. direction bit (from axis 0 to 3)	`dirAmp`
From 8 to 11	Amp. validation bit (from axis 0 to 3)	`inhAmp`, `WatchDog_it_` and `genWatchDog`
From 12 to 15	(none)	(none)

Table 8.1: MPIOSM pins software dependencies

Chapter 9

Pulse width modulation generator

PWM, or Pulse Width Modulation, is a method of controlling the amount of power to a load without having to dissipate any power in the load driver. For instance, PWM signals can be used to drive speed controllers or motor amplifiers.

9.1 The genPWM generic macro

9.1.1 The INIT option

```
genPWM(INIT,PWMchannel,periodDivider)
```

Called with $1 equal to INIT, this macro initializes the MIOS Pulse Width Modulation Submodule. Identifier of the PWM channel to configure is passed using argument PWMchannel. PWM channels are numbered from 0 to 3 (for submodules MPWMSM0 to MPWMSM3) and from 4 to 7 (for submodules MPWMSM16 to MPWMSM19). Argument periodDivider is used for setting the period of the PWM signal generated by the given MPWMSM module. Period is calculated as follows:

$$T_{PWM} = \texttt{periodDivider} * \frac{2}{f_{SYS}}$$

Where T_{PWM} is the PWM signal period and f_{SYS} is the system clock frequency (here 40 MHz).

Here is an example of M4 macro-call used for initializing PWM submodule 1 with a PWM period of 50 μs:

```
> genPWM(INIT, 1, 1000)
```

Please refer to MPC555 User's manual [2] subsection 15.12.1 for further details.

9.1.2 The LOOP option

```
genPWM(LOOP,PWMchannel,periodDivider,?int)
```

Called with $1 equal to LOOP, this macro drive the given MPWMSM submodule output PWM signal. MPWMSM submodule number is passed using argument PWMchannel (between 0 and 7). Argument periodDivider is used for setting the period of the PWM signal generated (as described above in subsection 9.1.1). PWM pulse width is passed, as input, using the third argument (int). int is a 32-bit positive integer variable which values may be between 0 and +periodDivider (corresponding to pulse width from 0 to T_{PWM}).

Here is an example of M4 macro-call that drives, on channel 1, a PWM signal (of a 50μs period) which duty-cycle values is to be found into variable val:

```
> genPWM(LOOP, 1, 1000, val)
```

9.1.3 The END option

```
genPWM(END,PWMchannel,periodDivider)
```

Called with `$1` equal to `END`, this macro reset PWM channel which number and duty cycle are passed using arguments `PWMchannel` and `periodDivider`. It consistes of reseting any signal generation, by forcing PWM pulse width to zero.

Here is an example of M4 macro-call used for reseting PWM submodule 1, previously configurated with a PWM period of 50 μs:

```
> genPWM(INIT, 1, 1000)
```

9.2 The PWM user macro

```
PWM(PWMchannel,periodDivider,?pulseWidth)
```

This user macro implements PWM output signal writing. `PWMchannel` argument is used for passing the user number of the PWM channel to drive (available values are between 0 and 3, currently other channels are not available). `periodDivider` argument allow to specify the PWM signal period (refer to above section 9.1.1 for details about Period calculation). Finally, the third argument, `pulseWidth`, is used for passing as input the 32-bit integer variable containing the pulse width value of the desired PWM signal.

Depending on the SynDEx macro generation context (variable `MGC`), `PWM` macro call performs PWM channel initialization, PWM output signal writing (during main loop execution) and finally PWM channel reset (during application finalization step).

Here is a piece of M4 macro executive that write a PWM signal (of a 50μs period) on channel 3 (i.e. MPWMSM3 output pin), which duty-cycle value is found into variable `val`:.

```
> alloc_(int, val)
> main_
>   dnl Initialization step (MGC==INIT)
>   PWM(3, 1000)
>   loop_
>     dnl Loop execution (MGC==LOOP)
>     PWM(3, 1000, val)
>   endloop_
>   dnl Finalization step (MGC==END)
>   PWM(3, 1000)
> endmain_
```

9.3 The PWM_it_ user macro

```
PWM_it_(PWMchannel,periodDivider,?pulseWidth)
```

Same as `PWM` but executed under timer interrupt. Code generated for this macro is included into PIT handler. Thus `pulseWidth` 32-bit integer variable content is read asynchronously each time PIT trigs an interrupt.

Chapter 10

Fast quadrature decode TPU function (for incremental encoder support)

The fast quadrature decode function is a TPU input function that uses two channels to decode a pair of out-of-phase signals in order to increment or decrement a (position) counter. It is particularly useful for decoding position and direction information from a slotted encoder in motion control systems.

10.1 The `genTPU_FQD` generic macro

10.1.1 The `INIT` option

```
genTPU_FQD(INIT,encoderID)
```

Called with `$1` equal to `INIT`, this macro initializes TPU[1] functions for fast decoding incremental encoders quadrature. Argument `encoderID` is used for passing the incremental encoder identifier. Only incremental encoders numbered from 0 to 3 can be configured. Let us remark that performing such initialization also reset the position counter (i.e. after initialization, position counter is equal to 0). For more details about fast quadrature decoding, refer to Motorola application note TPUPN02/D [3].

Here is an example of M4 macro-call that initializes TPU functions for reading incremental encoder 3 position.

```
> genTPU_FQD(INIT, 3)
```

Refer to Motorola application note TPUPN02/D [3] section 8 and 9 for a detailled description of TPU fast quadrature decode initialization.

10.1.2 The `LOOP` option

```
genTPU_FQD(LOOP,encoderID,!int)
```

Called with `$1` equal to `LOOP`, this macro reads incremental encoder signals and performs fast quadrature decoding. Desired encoder identifier is passed using `encoderID` argument. Available encoders are numbered from 0 to 3. Incremental encoder position (counts read since initialization has been performed), resulting from quadrature decoding, is returned in a 32-bit integer output variable (passed using argument `int`).

[1] Time Processor Unit (refer to MPC555 User's manual [2] section 17 for further details

Refer to Motorola application note TPUPN02/D [3] section 8 and 9 for a detailled description of TPU fast quadrature decode algorithm.

Here is an example of M4 macro-call that reads incremental encoder position (with for instance encoder ID 3) and that returns the result into variable `res`:

```
> genTPU_FQD(LOOP, 3, res)
```

10.1.3 The DLOOP option

```
genTPU_FQD(DLOOP,encoderID,!int)
```

Called with `$1` equal to `DLOOP`, this macro reads incremental encoder signals and performs fast quadrature decoding. Desired encoder identifier is passed using `encoderID` argument. Available encoders are numbered from 0 to 3. Incremental encoder derived position (counts read since last macro call), resulting from quadrature decoding, is returned in a 32-bit integer output variable (passed using argument `int`).

Here with `DLOOP` option, we intentionally reset position counter after reading. Hence the returned value correspond to the number of encoder counts accumulated since last macro call.

Refer to Motorola application note TPUPN02/D [3] section 8 and 9 for a detailled description of TPU fast quadrature decode algorithm.

Here is an example of M4 macro-call that reads incremental encoder position (with for instance encoder ID 3) and that returns the result into variable `res`:

```
> genTPU_FQD(DLOOP, 3, res)
```

10.1.4 The END option

```
genTPU_FQD(END,encoderID)
```

Called with `$1` equal to `END`, this macro resets TPU fast quadrature functions. It disables the TPU FQD channel priority previously assigned to counting and decoding operations (refer to TPUPN02/D [3] section 5).

Here is an example of M4 macro-call that resets TPU FQD channel 0:

```
> genTPU_FQD(END, 0)
```

10.2 The TPU_FQD user macro

```
TPU_FQD(encoderID,!counter)
```

This user macro implements position reading operation for incremental encoders. Argument `encoderID` is used for passing ID of the desired encoder (possible encoder ID are 0, 1, 2 or 3). Encoder position resulting from fast quadrature decoding is a 16-bit signed integer (values between -32768 and +32767). Position value (counts read since initialization step) is aligned and stored into the 32-bit integer output variable passed using argument `counter`.

The returned encoder position corresponds to the number of encoder impulses counted since initialization step. Let us remark that when maximum position count is reached (i.e. +32767) counter overflows and switchs back to -32768. The same way, when minimun position count is reached (i.e. -32768) counter underflows and switchs to +32767.

In order to prevent from this overflow effect, an additional algorithm has been added. This way, the `TPU_FQD` macro return a 32-bits integer containing the counts read since initialization step. Finally, the range of the argument `counter` is between -4294967296 and +4294967295.

Depending on the SynDEx macro generation context (variable `MGC`), `TPU_FQD` macro call performs TPU FQD channel initialization, encoder position acquisition (during main loop execution) and finally TPU FQD channel reset (during application finalization step).

Here is a piece of M4 macro-call that reads position counter of incremental encoder 1 and returns it into variable `res`:

```
> alloc_(int, res)
> main_
>   dnl Initialization step (MGC==INIT)
>   TPU_FQD(1)
>   loop_
>     dnl Loop execution (MGC==LOOP)
>     TPU_FQD(1, res)
>   endloop_
>   dnl Finalization step (MGC==END)
>   TPU_FQD(1)
> endmain_
```

Warning: be careful not to use encoder with too many impulses per revolution. If the sum of used encoder signals frequencies is higher than the TPU one you will get badly decoded values. It is highly recommended to check encoder fast quadradure decode returned results before implementing axis control!

10.3 The `dTPU_FQD` user macro

```
dTPU_FQD(encoderID,!counter)
```

This user macro implements position reading operation for incremental encoders. Argument `encoderID` is used for passing ID of the desired encoder (possible encoder ID are 0, 1, 2 or 3). Encoder position resulting from fast quadrature decoding is a 16-bit signed integer (values between -32768 and +32767). Derived position (counts read since last macro call) value is aligned and stored into the 32-bit integer output variable passed using argument `counter`.

Depending on the SynDEx macro generation context (variable `MGC`), `dTPU_FQD` macro call performs TPU FQD channel initialization, encoder position acquisition (during main loop execution) and finally TPU FQD channel reset (during application finalization step).

Here is a piece of M4 macro-call that reads position counter of incremental encoder 1 and returns it into variable `res`:

```
> alloc_(int, res)
> main_
>   dnl Initialization step (MGC==INIT)
>   dTPU_FQD(1)
>   loop_
>     dnl Loop execution (MGC==LOOP)
>     dTPU_FQD(1, res)
>   endloop_
>   dnl Finalization step (MGC==END)
>   dTPU_FQD(1)
> endmain_
```

Warning: be careful not to use encoder with too many impulses per revolution. If the sum of used encoder signals frequencies is higher than the TPU one you will get badly decoded values. It is highly recommended to check encoder fast quadradure decode returned results before implementing axis control!

10.4 The TPU_FQD_it_ user macro

```
TPU_FQD_it_(encoderID,!counter)
```

Same as TPU_FQD but executed under timer interrupt. Code generated for this macro is included into PIT handler. Thus counter 32-bit integer variable content is updated asynchronously each time PIT trigs an interrupt.

10.5 The dTPU_FQD_it_ user macro

```
TPU_FQD_it_(encoderID,!counter)
```

Same as dTPU_FQD but executed under timer interrupt. Code generated for this macro is included into PIT handler. Thus counter 32-bit integer variable content is updated asynchronously each time PIT trigs an interrupt.

Chapter 11

Queued serial peripheral interface

The queued serial peripheral interface (QSPI) is used to communicate with external devices through a synchronous serial bus. The QSPI is fully compatible with SPI sys-tems found on other Motorola products, but has enhanced capabilities.

11.1 The `QSPI` generic macro

11.1.1 The `INIT` option

```
genQSPI(INIT)
```

Called with `$1` equal to `INIT`, this macro initializes the queued serial peripheral interface. Typically, this macro is used each time you want to implement support for a new SPI device. Indeed, it performs the convenient configurations required for data transmissions between the MPC555 QSPI module and external SPI devices. For instance, this macro has been used for implementing SPI absolute encoder support initialisation step (see `genSPIencoder` at subsection 17.1.1)

Here is an example of M4 macro-call that initializes QSPI module:

```
> genQSPI(INIT)
```

Refer to MPC555 User's manual [2] section 14.7 for further details about QSPI module initialization.

11.1.2 The `END` option

```
genQSPI(END)
```

Called with `$1` equal to `END`, this macro reset the queued serial peripheral interface. Typically, this macro is used when no more interaction with SPI devices is required. It resets all QSPI module configuration registers and then halts it. For instance, this macro has been used for implementing SPI absolute encoder support finalization step (see `genSPIencoder` at subsection 17.1.1)

Here is an example of M4 macro-call that resets QSPI module:

```
> genQSPI(END)
```

Refer to MPC555 User's manual [2] section 14.7 for further details about QSPI module reset.

Chapter 12

Serial port support (POLLING implementation)

MPC555 provides user with serial communication capabilities. It includes a dual, independent, serial communication interface (DSCI) that allows to communicate with external devices through an asynchronous serial bus. The two SCI modules are functionally equivalents. Several transfer rates, clocking, and interrupt-driven communication options are available at user level.

In the following sections, when user manipulates character strings, we assume that he respects the convention depicted on figure 12.1. That is to say that character strings are expected to be ended by a null character (of ASCII code 0). This way of proceeding is commonly used with the majority of operating systems or other serial systems dealing with strings.

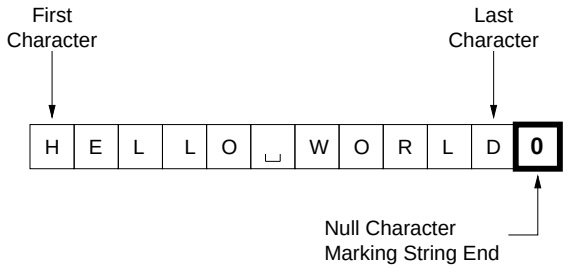

Figure 12.1: Description of a character string

In this chapter, serial communication interface (SCI) is implemented using polling method. By "polling" we mean that incoming data on the port are detected by infinite checking loop and not using SCI interrupts. The major part of the macros decribed here is not intended to be used by programmers. They are generally use for coding other convenient macros. User oriented macros, making use of SCI interrupts, are presented on chapter 13.

12.1 The `genSCIpoll` generic macro

12.1.1 The `INIT` option

```
genSCIpoll(INIT,SCIport,baud,n-bits,parity)
```

Called with `$1` equal to `INIT`, this macro initializes the SCI module which identifier is passed using `SCIport` argument. Two SCI modules are available SCI1 (i.e. `SCIport` = 1) and SCI2 (i.e. `SCIport` = 2). User can access various configurations through the third last arguments.

Argument `baud` allows to specify the desired serial communication baud rate. Please refer to table 12.1 that lists the possible values. Argument `n-bits` defines the desired SCI frame length: set to 10, `n-bits` configures SCI module in 10-bits SCI frame mode, while set to 11, `n-bits` configures SCI module in 11-bits SCI frame mode. Finally, `parity` argument is used for setting connexion parity: possible values are 0, 1 and 2, corresponding respectively to no parity, even parity and odd parity modes. All these modes can not be freely combined each other. Refer to table 12.2 that reports the possible configurations[1].

This macro also attach to the initialized serial port a round-robin buffer fully dedicated to receive operations. Round-robin buffer default size is fixed to 256 bytes but could be modified. For, user has to redefine the M4 variable `SCI_rxbuf_size` inside its own application dependent macro-executive file. For instance, the following definition sets buffer size to 1024 bytes:

```
> define(SCI_rxbuf_size,1024)
```

`baud` argument value	Actual baud rate in $bits.s^{-1}$)	Error (%)
1250000	1,250,000.00	0.00
115200	113,636.36	-1.37
57600	56,818.18	-1.36
38400	37,878.79	-1.36
32768	32,894.74	0.39
28800	29,069.77	0.94
19200	19,230.77	0.16
14400	14,367.81	-0.22
9600	9,615.38	0.16
4800	4,807.69	0.16
2400	2,399.23	-0.03
1200	1,199.62	-0.03
600	600.09	0.02
300	299.98	-0.01

Table 12.1: List of the baud rates and related percent error

SCI frame length	Parity	Resulting serial frame format
10 bits (10)	0 none (0)	8 data bits and 2 stop bits
10 bits (10)	even (1)	7 data bits, 1 parity bit and 2 stop bits
10 bits (10)	odd (2)	7 data bits, 1 parity bit and 2 stop bits
11 bits (11)	none (0)	9 data bits and 2 stop bits
11 bits (11)	even (1)	8 data bits, 1 parity bit and 2 stop bits
11 bits (11)	odd (2)	8 data bits, 1 parity bit and 2 stop bits

Table 12.2: List of the possible SCI module configurations

Here is an example of M4 macro-call that initializes SCI module 1 with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> genSCIpoll(INIT,1,4800,10,2)
```

Further details about SCI modules configuration can be found in MPC555 User's manual [2] sections 14.8.2 and 14.8.3.

[1] Let us note that the most uncommon configuration mode (of frame format 9 data bits and 2 stop bits) has not been tested.

12.1.2 The RD option

```
genSCIpoll(RD,SCIport,baud,n-bits,parity)
```

alled with `$1` equal to `RD`, this macro allows to fill the round-robin buffer dedicated to current serial port receive operations.

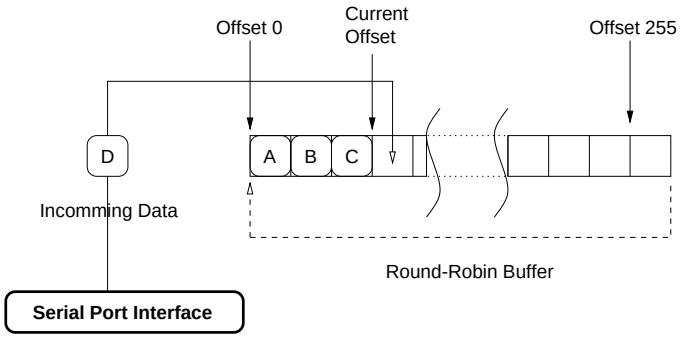

Figure 12.2: Serial port round-robin buffer

Initially, round-robin buffer pointer is located at offset 0. Each incoming character detected on the current serial port is immediately stored into the round-rubin buffer. Once the character has been stored at the pointed buffer address, round-robin buffer pointer is incremented awaiting next character. When buffer pointer reach buffer end (by default offset 255), next incrementation make it switch back to offset 0. Warning: if data stored has not been read when the buffer overflows, they are inevitablly overwritten (see figure 12.2). For instance, let us assume that unread data location begins at offset n, that buffer pointer has already switched back to offset 0 and that it currently moves toward offset n storing incoming characters. Then, data begining at offset n, may be accessed while buffer pointer differs from offset n address. As soon as buffer pointer refers to offset n or further, data are overwritten and may not be recovered.

Details about `SCIport`, tt baud, `n-bits` and `parity` arguments settings are given above at subsection 12.1.1.

Here is an example of M4 macro-call that fills the round-robin buffer allocated for receive operations on serial port 1. Serial port 1 is assumed to be configurated with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> genSCIpoll(RD,1,4800,10,2)
```

12.1.3 The WRC option

```
genSCIpoll(WRC,SCIport,baud,n-bits,parity,?int)
```

Called with `$1` equal to `WRC`, this macro can be used for testing serial port (of ID `SCIport`) configuration. Configuration parameters passed to the macros are used the same way as they are in the macros described above. The last input argument `int` indicates the decimal value of the ASCII character intended to be written on the serial port. Note, that `int` is a 32-bits integer.

Here is an example of M4 macro-call that writes an 'A' on serial port 1. As before, in this example, serial port 1 is assumed to be configurated with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> alloc_(int,val); li r0,65; B(stw r0,val) dnl 65 is the A ASCII decimal code.
> genSCIpoll(WRC,1,4800,10,2,val)
```

12.1.4 The WRB option

```
genSCIpoll(WRB,SCIport,baud,n-bits,parity)
```

Called with $1 equal to WRB, this macro is used for writing, on serial port SCIport, the last received character string, stored into its allocated round-robin buffer. This macro is also a demo macro aiming to control if incoming character strings are correctly read and stored into the round-robin buffer.

Here is an example of M4 macro-call that writes on serial port 1 the last character string received on it. As before, in this example, serial port 1 is assumed to be configurated with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> genSCIpoll(WRB,1,4800,10,2)
```

12.1.5 The STB option

```
genSCIpoll(STB,SCIport,baud,n-bits,parity,!buf)
```

Called with $1 equal to STB, this macro allows to copy the last character string received on serial port SCIport into a buffer. The buffer to fill is passed using the last argument buf.

Be careful. Let us remark that no verification is done at that level: thus, the user buffer is expected to be large enough to receive the character string. Otherwise, contiguous data, following user buffer buf, may be affected.

Here is an example of M4 macro-call that fills a 256-bytes user buffer (named my_buf) with the last character string received on serial port 1. In this example, serial port 1 is assumed to be configurated with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> dnl Allocate the 256-characters buffer my_buf.
> alloc_(char,my_buf,256)
> genSCIpoll(STB,1,4800,10,2,my_buf)
```

12.1.6 The STN option

```
genSCIpoll(STN,SCIport,baud,n-bits,parity,!buf,n_char)
```

Called with $1 equal to STN, this macro allows to copy, into a user buffer, a given number of characters from the serial port SCIport round-robin buffer. This copy starts from the first unread character (available in serial port SCIport the round-robin buffer) and stops when it reaches the given amount of characters. The buffer to fill is passed using the 6th input argument buf, while the number of characters to copy is passed using the last argument n_char. Argument n_char is expected to be a positive integer. If n_char is greater than the number of characters currently available in the round-robin buffer, buf stays unchanged.

Be careful. Let us remark that no verification is done at that level: thus, the user buffer is expected to be large enough to receive the specified number of character. Otherwise, contiguous data, following user buffer buf, may be affected.

Here is an example of M4 macro-call that fills a 256-bytes user buffer (named my_buf) with the last 10 characters received on serial port 1. In this example, serial port 1 is assumed to be configurated with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> dnl Allocate the 256-characters buffer my_buf.
> alloc_(char,my_buf,256)
> genSCIpoll(STN,1,4800,10,2,my_buf,10)
```

12.1.7 The WRA option

```
genSCIpoll(WRA,SCIport,baud,n-bits,parity,?int)
```

Called with $1 equal to WRA, this macro is used for writing, on serial port SCIport, the character string, stored into the buffer which address is passed using the last input argument int. To this aim, int is expected to be 32-bit integer variable containing the buffer address. This macro is mainly use for internal coding or for coding other serial port operation.

Be careful. Let us remark that no verification is done at that level: if int contains a bad value (i.e. that corresponds to no physical memory or that indicates an other location) your program may crash or send through the serial port unexpected data.

Here is an example of M4 macro-call that writes on serial port 1 the character string contained in a buffer which address is 0x003F993. As before, in this example, serial port 1 is assumed to be configurated with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> dnl Allocate integer variable var
> alloc_(int,var)
> dnl Now set var to buffer address.
> li r0,0x003F993; B(stw r0, var)
> genSCIpoll(WRA,1,4800,10,2,var)
```

Warning: be sure that the string you want to write is ended by a null character (of ASCII code 0). The null character, well marking string end, is used for stopping string reading process. Thus if no null character is to be found, macro will read unexpected character values over string end until it encounters an other zero in memory. Refer to figure 12.1.

12.1.8 The WRS option

```
genSCIpoll(WRS,SCIport,baud,n-bits,parity,?char)
```

Called with $1 equal to WRS, this macro is used for writing, on serial port SCIport, the character string, stored into a given buffer. This buffer is passed using the last input argument char, which is the the first character of the string. Be careful. Let us remark that no verification is done at that level: if char contains a bad value (i.e. that corresponds to no physical memory or that indicates an other location) your program may crash or send through the serial port unexpected data.

Let us focus on the following example. Let my_buf be a character buffer previously filled using a genSCIpoll(STB,...) macro-call. Here is the way to use WRS option for writing on serial port 1 the string contained in my_buf. As before, in this example, serial port 1 is configurated with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> dnl Allocate the 256-characters buffer my_buf.
> alloc_(char,my_buf,256)
> dnl Fill it with STB option.
> genSCIpoll(STB,1,4800,10,2,my_buf)
> dnl Now, write my_buf content on serial port 1.
> genSCIpoll(WRS,1,4800,10,2,my_buf)
```

Warning: be sure that the string you want to write is ended by a null character (of ASCII code 0). The null character, well marking string end, is used for stopping string reading process. Thus if no null character is to be found, macro will read unexpected character values over string end until it encounters an other zero in memory. Refer to figure 12.1.

12.1.9 The WRN option

```
genSCIpoll(WRN,SCIport,baud,n-bits,parity,?char,?n_char)
```

Called with $1 equal to WRN, this macro is used for writing, on serial port SCIport, n_char characters of the string, stored into the given buffer. This buffer is passed using the 6th input argument char, which is the the first character of the string. While, n_char argument is expected to be a positive integer. If n_char is greater than the number of characters currently available in the round-robin buffer, this call will write on serial port unexpected data

Be careful. Let us remark that no verification is done at that level: if char contains a bad value (i.e. that corresponds to no physical memory or that indicates an other location) your program may crash or send through the serial port unexpected data.

Let us focus on the following example. Let my_buf be a character buffer previously filled using a genSCIpoll(STB,...) macro-call. Here is the way to use WRN option for writing on serial port 1, 10 characters of the the string contained in my_buf. As before, in this example, serial port 1 is configurated with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> dnl Allocate the 256-characters buffer my_buf.
> alloc_(char,my_buf,256)
> dnl Fill it with STB option.
> genSCIpoll(STB,1,4800,10,2,my_buf)
> dnl Now, write my_buf content on serial port 1.
> genSCIpoll(WRN,1,4800,10,2,my_buf,10)
```

12.1.10 The END option

```
genSCIpoll(END,SCIport)
```

Called with $1 equal to END this macro resets the SCI module which identifier is passed using SCIport argument. Basically, it sets the convenient SCI control register (SCCxR1) to default value. As before, possible SCIport values are 1 or 2, respectively for serial ports 1 or 2.

Here is an example of M4 macro-call that reset serial port 1:

```
> genSCIpoll(END,1)
```

Refer to MPC555 User's manual [2] section 14.8.3 for further details).

Chapter 13

Serial port support (interrupt handling implementation)

This chapter introduces user oriented macros for manipulating MPC555 serial communication interface (SCI). These macros implement SCI support using interrupt handling method. This technique allows to detect incoming data on serial port by triggering SCI module interrupt, what saves a lot of computation band-width compared with polling techniques.

13.1 The genSCI_it_ generic macro

13.1.1 The INIT option

```
genSCI_it_(INIT,SCIport,baud,n-bits,parity)
```

Called with `$1` equal to `INIT` this macro initializes the serial port which identifier is passed using argument `SCIport` (possible values are 1 and 2 for respectively serial ports 1 and 2). The following arguments (`baud`, `n-bits` and `parity`) are used for serial port configuration purpose. Please refer to subsection 12.1.1 for details about how to set these parameters.

This macros also installs the corresponding SCI module interrupt handler. This handler is in charge of reading on the serial port (of ID `SCIport`) incoming data, each time an SCI interrupt is detected. Once read, handler code appends the data to the round-robin buffer content. Hence, data reading operations are not done by user commands, but perfomed asynchronously and automatically by the handler. Thus, user does not have to take care about incoming data, he just has to read the round-robin buffer content. Please refer to subsection 13.2 that explains how user can access strings read by the handler.

Here is an example of M4 macro-call that performs serial port 1 initialization, interrupt handler installation and allocation for its proper round-robin buffer. Here, serial port 1 is configurated with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> genSCI_it_(INIT,1,4800,10,2)
```

13.1.2 The END option

```
genSCI_it_(INIT,SCIport,baud,n-bits,parity)
```

Called with `$1` equal to `END` this macro terminates activity of the serial port (of ID `SCIport`) and, consequently, activity of the corresponding interrupt handler.

Let us remark that this macro call does not reset the round-robin buffer attached to the serial port. Hence, data stored into the buffer still may be accessed even after the serial port has been closed. Anyway, if a new initialization is done (following such reset step) round-robin buffer pointer will be set back to buffer offset 0.

Here is an example of M4 macro-call that performs serial port 1 reset.

```
> genSCI_it_(END,1,4800,10,2)
```

Let us note that, `genSCI_it_` macro only aims to be used for coding other macros, but not to be used in user programs. For instance, this macros is involved in the `SCI_it_gets` user macro implementation. Refer to section 13.2 for more details.

13.2 The `SCI_it_gets` user macro

```
SCI_it_gets(SCIport,baud,n-bits,parity,!buf)
```

This user macro implements serial port operations for reading incoming character strings. User can specify the desired serial port using `SCIport` argument (of value 1 or 2). The following arguments (`baud`, `n-bits` and `parity`) are used for serial port configuration purpose. Please refer to subsection 12.1.1 for details about how to set these parameters. Incoming character strings are first stored into the serial port associated round-robin buffer and then copied into the user buffer `buf` (passed as the last argument).

Depending on the macro generation context (variable MGC), `SCI_it_gets` macro call performs serial port initialization, character strings acquisitions (during main loop execution) and finally serial port reset (during application finalization step).

Here is a piece of M4 macro executive that fills user buffer `buf` with each new incomming character strings received on serial port 1. In this example, serial port 1 is assumed to be configurated with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> alloc_(char,buf,256)
> main_
>   dnl Initialization step (MGC==INIT)
>   SCI_it_gets(1,4800,10,2)
>   loop_
>     dnl Loop execution (MGC==LOOP)
>     SCI_it_gets(1,4800,10,2,buf)
>   endloop_
>   dnl Finalization step (MGC==END)
>   SCI_it_gets(1,4800,10,2)
> endmain_
```

13.3 Note for users

Let us remark that incoming character strings are not always ended by the null character (ASCII code 0). SCI macros implementing reading operation have to be informed of the character code used for marking string end. To this aim, user is expected to define the macro `SCI1ENDCHR` (resp. `SCI2ENDCHR`) each time he invokes reading operations on SCI1 (resp. SCI2) serial port. This macro should contain the decimal ASCII value of the character marking string end. Defining this macro may be done as follow:

```
> dnl Character of ASCII code (10)d marks SCI2 incoming strings end
> define('SCI2ENDCHR',10)
```

13.4 The SCI_puts user macro

```
SCI_puts(SCIport,baud,n-bits,parity,?buf)
```

This user macro implements operations for writing character strings on serial port. User can specify the desired serial port using `SCIport` argument (of value 1 or 2). The following arguments (`baud`, `n-bits` and `parity`) are used for serial port configuration purpose. Please refer to subsection 12.1.1 for details about how to set these parameters. The string to be sent should be located in the user buffer `buf` (passed as the last argument). As mentionned in the previous chapter introduction, do not forget to end strings with a null character (of ASCII code 0).

Depending on the macro generation context (variable MGC), `SCI_puts` macro call performs serial port initialization, character strings send (during main loop execution) and finally serial port reset (during application finalization step).

Here is a piece of M4 macro executive that sends user buffer `buf` content on serial port 1. In this example, serial port 1 is assumed to be configurated with a 4800 baud rate, odd parity and 7 data bits serial frame format:

```
> alloc_(char,buf,256)
> main_
>   dnl Initialization step (MGC==INIT)
>   SCI_puts(1,4800,10,2)
>   loop_
>     dnl Loop execution (MGC==LOOP)
>     SCI_puts(1,4800,10,2,buf)
>   endloop_
>   dnl Finalization step (MGC==END)
>   SCI_puts(1,4800,10,2)
> endmain_
```

Very important remark

When using one (or more) `SCI_puts` macro call(s), you should be very careful of the time delay required for realizing all the serial transmissions. In other words, serial transmissions time delay must respect the real-time constraints of your application. For instance, considering an application which real-time period is 10 ms, using a 9600 bauds, 8 data bits and no parity serial port setting, you can not send more than 9 characters each period. If you do not respect this, your application will crash.

Part III

Robosoft board specific macros

Chapter 14

Robosoft board initializations

As a proprietary control system, Robosoft board embeds no commercial operating system. Hence, board initialization step correspond to no standard procedure. For, specific settings have to be performed before any software execution, both insuring the board to be an a proper state and enabling all board features.

14.1 The `main_ini_` macro

```
main_ini_()
```

This macro should never be used by programmers. It is an internal SynDEx macro, automatically called when generating MPC555 executive. This macro is presented here only for information. Indeed Robosoft board settings may help expert programmers to get a deep understanding of both the board and SynDEx MPC555 kernel. Reading this section is not of interest for other programmers.

First, MPC555 system configuration is done setting SIU Module Configuration Register (SIUMCR, refer to MPC555 User's manual [2] subsection 6.13.1.1). SC (SIUMCR bits 17 to 18) is set to 1, what activates "multiple chip, 16-bits port size" mode (refer to [2] Table 6-9 for corresponding pins configuration). MLRC (SIUMCR 20 to 21) is set to 2, what makes IRQ pin configured as general-purpose I/O (refer to [2] Table 6-10 for corresponding pins configuration).

Setting the external master control register (EMCR, refer to [2] subsection 6.13.1.3), we select the external master modes and determine the internal bus attributes for external-to-internal accesses. Here we keep EMCR reset value except for SIZE (EMCR bit 21 to 20) we set to 2, corresponding to an half-word (2 bytes) internal bus attributes configuration.

What follows is used for controlling the memory bank 1 controller. For, we have to set BR1 register (refer to [2] subsection 10.8.3). In order to indicate that BR and OR are valid, we set V (BR1 bit 31) to 1. With BA (BR1 bits 0 to 15) we configure memory bank 1 base address to 0xFFF0. BI (BR1 bit 30) set to 1 indicates memory bank 1 does not support burst accesses. Next, WEBS (BR1 bit 26) set to 1, makes the he $\overline{WE}/\overline{BE}$ pads operate as $\overline{BE}$. Finally, PS (BR1 bit 20 to 21) set to 2, configures port size to 16-bit.

In association with BR1 register, the memory controller option registers 1 (OR1) has to be set (refer to [2] subsection 10.8.4). AM (OR1 bits 0 to 15) set to 0xFFFE, allows masking of any corresponding bits in the associated base register. Masking the address bits independently allows external devices of different size address ranges to be used. Any clear bit masks the corresponding address bit. Any set bit causes the corresponding address bit to be used in comparison with the address pins. Next, ATM (OR1 bits 17 to 19) is set to 0 allows to ignore address type codes as part of the address comparison. We set CSNT (OR1 bit 20) to 0, forcing $\overline{CS}/\overline{WE}$ to be negated normally. Setting ACS (OR1 bits 21 to 22) to 0 indicates that $\overline{CS}$ is asserted at the same time that the address lines are valid. EHTR (OR1 bit 23) is set to 0, configuring memory controller for

generating normal timing. Finally, SCY (OR1 bit 24 to 27) is set to 0, specifying that 0 wait states should be inserted in the single cycle, or in the first beat of a burst, when the GPCM handles the external memory access.

In order to enable the MPC555 time base and decrementer, we set the time base control and status register (TBSCR, refer to [2] subsection 6.13.4.4). To this aim, TBE (TBSCR bit 15) is set to 1.

Now, we need to indicate IMB frequency. For, we have to set the pad module configuration register (PDMCR, refer to [2] subsection 2.4.2). SLRC0 (PDMCR bit 0) set to 1, configures normal slew rate for TPU, QADC, USIU (SGPIO). SLRC1 (PDMCR bit 1) set to 1, configures normal slew rate for QSPI and TouCAN modules. SLRC2 (PDMCR bit 2) set to 1, configures normal slew rate for QSCI. SLRC3 (PDMCR bit 3) set to 1, configures normal slew rate for MIOS. Finally, PRDS (PDMCR bit 6) is set to 0, enabling pull-up/pull-down devices.

Last, still remains the PLL configuration. For, we have to set the PLL, low-power, and reset-control register (PLPRCR, refer to [2] subsection 8.12.2). We need to keep power on time reset values except for the following bits: MF (PLPRCR bits 0 to 11) and TEXPS (PLPRCR bit 17). MF is set to 9, dividing the feedback signal by 10 for the PLL the phase comparator. TEXPS is set to 0, indicating timer expired status bit is negated in deep-sleep mode.

Chapter 15

Digital to analog converter

Digital to analog conversion is a process in which a binary signal (defined over two states) are converted into signals having a theoretically infinite number of states (analog). Basically, digital to analog conversion is the opposite of analog to digital conversion. In most cases, if an analog to digital converter (ADC) is placed after a DAC, the digital signal output is identical to the digital signal input. Also, in most instances when a DAC is placed after an ADC, the analog signal output is identical to the analog signal input.

15.1 The genDAC generic macro

15.1.1 The INIT option

```
genDAC(INIT)
```

Called with `$1` equal to `INIT`, this macro initializes the Robosoft board DAC (Digital-to-Analog Converter) chip. DAC chip actually in use is from Analog Device and is referenced as DAC8420 (refer to DAC8420 datasheet [4] for further details). At initialization step, let us remark that all DAC channels are initialized regardless to the channels that will really be used.

As a serial converter, DAC8420 access is provided by MIOS parallel port I/O submodule (MPIOSM, refer to MPC555 User's manual [2]). MPIOSM pins D0, D1 and D2 are initialized for this. Pin D0 is dedicated to serial data transfer, pin D1 is dedicated to clock signal generation and pin D2 used for loading data into the DAC8420.

Here is an example of M4 macro-call that initializes DAC chip:

```
> genDAC(INIT)
```

15.1.2 The LOOP option

```
genDAC(LOOP, DACchannel, ?int)
```

Called with `$1` equal to `LOOP`, this macro allows to drive an analog signal on a given DAC channel (which number is passed using argument `DACchannel`). Accessible DAC channels are numbered from 0 to 3. Correspondance between analog channels numbers and DAC8420 hardware pins is given in table 15.1. Also refer to DAC8420 datasheet [4] page 6 for pin description and location. The last input argument (`int`) is used for passing the analog signal level. `int` is a 32-bit integer which values are expected to be from 0 to 0x0FFF (i.e. from 0 to 4095), corresponding to analog output voltages from -10V to +10V.

Here is an example of M4 macro-call that drives, on DAC channel 1, a continuous signal, which values is to be found into variable `val`:

DAC channel number	Reference to DAC8420 chip
0	VOUTA pin 7
1	VOUTB pin 6
2	VOUTC pin 3
3	VOUTD pin 2

Table 15.1: DAC Channel number assignments and pin designations

```
> genDAC(LOOP,1,val)
```

An exhaustive description of serialized data transfers needed for driving the DAC is given on DAC8420 datasheet [4] pages 6 and 15. You can also find this in the `RSB.m4x` macro executive source code in the section named "Digital to analog converter" of the "Standard on-board I/O macros" part.

15.1.3 The `END` option

```
genDAC(END, DACchannel)
```

Called with `$1` equal to `END`, this macro reset analog output channel which number is passed using argument `DACchannel` (of value from 0 to 3). It consistes of reseting any signal generation, by forcing the selected analog output to zero.

Here is an example of M4 macro-call that resets DAC channel 1:

```
> genDAC(END,1)
```

15.2 The `DAC` user macro

```
DAC(DACchannel, ?analogValue)
```

This user macro implements DAC writing operations. Argument `DACchannel` is used for selecting the desired DAC channel (numbered from 0 to 3). Voltage value, of the signal to be driven on the selected DAC channel, is passed using the 32-bit integer input variable `analogValue`.

Depending on the SynDEx macro generation context (variable MGC), `DAC` macro call performs DAC channel initialization, DAC output signal writing (during main loop execution) and finally DAC channel reset (during application finalization step).

Here is a piece of M4 macro executive that writes on DAC channel 1 a continuous signal, which voltage value is to be found into variable `val`:

```
> alloc_(int, val)
> main_
>   dnl Initialization step (MGC==INIT)
>   DAC(1)
>   loop_
>     dnl Loop execution (MGC==LOOP)
>     DAC(1, val)
>   endloop_
>   dnl Finalization step (MGC==END)
>   DAC(1)
> endmain_
```

15.3 The DAC_it_ user macro

```
DAC_it_(DACchannel, ?analogValue)
```

Same as DAC macro, but executed under timer interrupt. Code generated for this macro is included into PIT handler. Thus analogValue 32-bit integer variable content is read asynchronously each time PIT trigs an interrupt.

Chapter 16

On-board LED interface

16.1 The genLED generic macro

16.1.1 The INIT option

```
genLED(INIT)
```

Called with $1 equal to INIT, this macro initializes the Robosoft board LED. Connected to QSPI module pin PCS1 (refer to MPC555 User's manual [2] Table 14-8), this macro performs QSPI module initialization, by calling the QSPI macro (refer to subsection 11.1.1). At that step, LED initialization automalically reset LED state, i.e. it sets LED off.

Here is an example of M4 macro-call that initializes the on-board LED:

```
> genLED(INIT)
```

16.1.2 The ON option

```
genLED(ON)
```

Called with $1 equal to ON, this macro set the Robosoft board LED on.

Here is an example of M4 macro-call that sets the on-board LED on:

```
> genLED(ON)
```

16.1.3 The OFF option

```
genLED(OFF)
```

Called with $1 equal to OFF, this macro set the Robosoft board LED off.

Here is an example of M4 macro-call that sets the on-board LED off:

```
> genLED(OFF)
```

16.1.4 The LOOP option

```
genLED(LOOP, ?bool)
```

Called with `$1` equal to `LOOP`, this macro set the Robosoft board LED on or off, depending on the value of the 8-bit boolean input variable `bool`. If `bool` value is 0 LED is set off, else LED is set on.

Here is an example of M4 macro-call that sets the on-board LED, depending on boolean variable `val` value:

```
> genLED(LOOP, val)
```

16.1.5 The `END` option

```
genLED(END)
```

Called with `$1` equal to `END`, this macro resets the Robosoft board LED. At that step, LED is set LED off. This macro-call also reset QSPI module by calling the `QSPI` macro (refer to subsection 11.1.2).

Here is an example of M4 macro-call that rests the on-board LED:

```
> genLED(END)
```

16.2 The `LED` user macro

```
LED(?bool)
```

This user macro implements LED setting operations. Input argument `bool` is used for passing name of the 8-bit boolean variable that contains LED state.

Depending on the SynDEx macro generation context (variable MGC), `LED` macro call performs on-board LED initialization, LED state setting (during main loop execution) and finally on-board LED reset (during application finalization step).

Here is a piece of M4 macro executive that handle the on-board LED, which state value is to be found into variable `val`:

```
> alloc_(bool, val)
> main_
>   dnl Initialization step (MGC==INIT)
>   LED()
>   loop_
>     dnl Loop execution (MGC==LOOP)
>     LED(val)
>   endloop_
>   dnl Finalization step (MGC==END)
>   LED()
> endmain_
```

16.3 The `LED_it_` user macro

```
LED_it_(?bool)
```

Same as `LED` macro, but executed under timer interrupt. Code generated for this macro is included into PIT handler. Thus `bool` 8-bit boolean variable content is read asynchronously each time PIT trigs an interrupt.

Chapter 17

SPI absolute encoder

17.1 The genSPIencoder generic macro

17.1.1 The INIT option

```
genSPIencoder(INIT)
```

Called with $1 equal to INIT, this macro initializes the Robosoft board absolute encoder interface. Connected to QSPI module pin MISO (refer to MPC555 User's manual [2] Table 14-8), this macro performs QSPI module initialization, by calling the QSPI macro (refer to subsection 11.1.1).

Here is an example of M4 macro-call that initializes the on-board absolute encoder interface:

```
> genSPIencoder(INIT)
```

17.1.2 The LOOP option

```
genSPIencoder(LOOP,!int)
```

Called with $1 equal to LOOP, this macro read absolute encoder position. Returned value is copied into the 32-bit integer variable which name is passed using argument int. Range of values returned by this macro depends on the type of encoder you are using. For instance on CyCab and Robucar Robosoft products, returned encoder position are between 0 and 8191 (corresponding to the number of impulses got during a complete revolution).

Information related to the data serial transfer protocol, between MPC555 and absolute encoder device, can be found in MPC555 User's manual [2] section 14.7.

Here is an example of M4 macro-call that reads absolute encoder position and that returns result into variable val:

```
genSPIencoder(LOOP, val)
```

17.1.3 The END option

```
genSPIencoder(END)
```

Called with $1 equal to END, this macro resets the Robosoft board absolute encoder interface. This macro also performs QSPI module reset, by calling the QSPI macro (refer to subsection 11.1.2).

17.2 The `SPIencoder` user macro

```
SPIencoder(!int)
```

This user macro implements absolute encoder reading operations. Input argument `int` is used for passing name of the 32-bit integer variable in which encoder position is returned. As mentionned above, Range of values returned by `SPIencoder` depends on the type of encoder you are using. For instance on CyCab and Robucar Robosoft products, returned encoder position are between 0 and 8191 (corresponding to the number of impulses got during a complete revolution).

Depending on the SynDEx macro generation context (variable MGC), `SPIencoder` macro call performs SPI encoder initialization, SPI encoder position reading (during main loop execution) and finally SPI encoder reset (during application finalization step).

Here is a piece of M4 macro executive that reads absolute encoder position and returns it into variable `val`:

```
> alloc_(int, val)
> main_
>   dnl Initialization step (MGC==INIT)
>   SPIencoder()
>   loop_
>     dnl Loop execution (MGC==LOOP)
>     SPIencoder(val)
>   endloop_
>   dnl Finalization step (MGC==END)
>   SPIencoder()
> endmain_
```

17.3 The `SPIencoder_it_` user macro

```
SPIencoder_it_(!int)
```

Same as `SPIencoder` macro, but executed under timer interrupt. Code generated for this macro is included into PIT handler. Thus `int` 32-bit boolean variable content is read asynchronously each time PIT trigs an interrupt.

Chapter 18

Power amplifier direction setting

18.1 The `dirAmp` generic macro

18.1.1 The `INI` option

```
dirAmp(DEF, AmpID)
```

Called with `$1` equal to `INI` this macro reset all motor power amplifiers directions. This macro-call should normally be invoked into the `INIT` context when coding a user macro).

Here is an example of M4 macro-call that reset all power amplifiers directions:

```
> dirAmp(INI)
```

18.1.2 The `DEF` option

```
dirAmp(DEF, AmpID)
```

Called with `$1` equal to `DEF` this macro set motor power amplifier default direction. Power amplifier ID is passed using argument `AmpID`: possible values for `AmpID` are 0, 1, 2 or 3.

Here is an example of M4 macro-call that set power amplifier default direction for motor, which ID is 2:

```
> dirAmp(DEF, 2)
```

18.1.3 The `INV` option

```
dirAmp(INV, AmpID)
```

Called with `$1` equal to `INV` this macro set motor power amplifier opposite direction. Power amplifier ID is passed using argument `AmpID`: possible values for `AmpID` are 0, 1, 2 or 3.

Here is an example of M4 macro-call that set power amplifier opposite direction for motor, which ID is 2:

```
> dirAmp(INV, 2)
```

18.2 Note for users

This macro may be needed when you intend to drive motors. For instance, some PWM power amplifiers provide user with a direction entry. With such devices, the full PWM duty-cycle range is available for driving motor in a single direction. Thus, direction bit is used for switching motor from one direction to an other (independently of the PWM signal value).

Last, let us note that direction signals, for motors from 0 to 3, are driven through MIOS parallel port I/O submodule pins from 4 to 7. Please refer to MPC555 User's manual [2] section 15.13 for further details.

Chapter 19

Motor power amplifier validation

19.1 The inhAmp generic macro

19.1.1 The ENA option

```
inhAmp(ENA, AmpID)
```

Called with $1 equal to ENA this macro enable motor power amplifier which ID is passed using argument AmpID. Possible values for AmpID are 0, 1, 2 or 3. Thus, this macro is said to switch on the motor power amplifier inhibition signal.

Let us note that this macro is absolutly needed when you want to drive a motor. For security matters, we designed our board in a way that simply driving a PWM or analog signal onto a power amplifier is not sufficent. User has to explicitly validate the power amplifier. By the way, non-intentional signal activity on power amplifier port may not lead to any motor moves.

In major cases, motors embedded in Robosoft products are equiped with electrically commanded brakes. Most of the time, inhibition signal (set using this macro) is coupled with motor brake. Hence, if inhibition signal validate power amplifier, motor brake is released, else, if inhibition signal is not active, motor brake is engaged. Thus, inhAmp(ENA) macro-call may be used for releasing motor brake.

Here is an example of M4 macro-call that validates power amplifier for motor, which ID is 2, and, consequently, releases its brake:

```
> inhAmp(ENA, 2)
```

19.1.2 The DIS option

```
inhAmp(DIS, AmpID)
```

Called with $1 equal to DIS this macro disable motor power amplifier which ID is passed using argument AmpID. Possible values for AmpID are 0, 1, 2 or 3. Thus, this macro is said to switch off the motor power amplifier inhibition signal.

Let us remark, that most often, motors embedded in Robosoft products are equiped with electrically commanded brakes. Inhibition signal (set using this macro) is generally coupled with motor brake. Hence, if inhibition signal validate power amplifier, motor brake is released, else, if inhibition signal is not active, motor brake is engaged. Thus, inhAmp(DIS) macro-call may be used for engaging motor brake.

Inhibition signals, for motors from 0 to 3, are driven through MIOS parallel port I/O submodule pins from 8 to 11. Please refer to MPC555 User's manual [2] section 15.13 for further details.

Here is an example of M4 macro-call that switchs off power amplifier for motor, which ID is 2, and, consequently, engages its brake:

```
> inhAmp(DIS, 2)
```

19.2 Note for users

Let us remark that power amplifier validation state can be controlled (using `inhAmp` macro) only if Robosoft software watchdog is activated (cf. chapter 20 for watchdog macros description).

Inhibition signals, for motors from 0 to 3, are driven through MIOS parallel port I/O sub-module pins from 8 to 11. Please refer to MPC555 User's manual [2] section 15.13 for further details.

Chapter 20

Watch dog

For security purposes, Robosoft adds an hardware watch-dog mechanism on its board. This watch-dog aims to stop motors activity when, for any reasons, running software crashes. Watch-dog is implemented as an hadware time-out. If it is not refreshed periodically, watch-dog mechanism drives an output signal that forces motor power amplifiers validation to go down. By this way the controlled system stays in a safe state, with no risk to perform hazardous motions.

20.1 The `genWatchDog` generic macro

20.1.1 The `INIT` option

```
genWatchDog(INIT)
```

Called with `$1` equal to `INIT`, this macro initializes the Robosoft on-board watch-dog.

The watch-dog is physically connected to the chip select pin of the MPC555 memory bank located at 0x0FF00000. A single access to this memory bank will generate a chip select and will assert the watch-dog logic. While the watch-dog is refreshed, the MPC555 board allow motor control. When the watch-dog is no longer refreshed, the MPC555 board disable motor control. Thus as it shadows user inhibition signal effects, once down, watch-dog automalically engages motors brakes.

Watch-dog logic is realized with a Philips one-shot multivibrator: 74HC4538. Refer to the 74HC4538 data-sheet [5] for more information about it. We use this logic as a retriggerable monostable circuitry (for refreshing watch-dog). Actually, 74HC4538 chip is equiped of a $200k\Omega$ R_X resistor and a $100nF$ C_X capacitor (refer to the 74HC4538 data-sheet [5] page 10 for application information). This set watch-dog refresh period to $20ms$. If, for any reason, user program exceeds this period, watch-dog goes down and disable motor control as described above.

Here is an example of M4 macro-call that initializes the Robosoft on-board watch-dog:

```
> genWatchDog(INIT)
```

Refer to MPC555 User's manual [2] subsections 10.8.3 and 10.8.4 for further details about the MPC555 memory controller.

20.1.2 The `LOOP` option

```
genWatchDog(LOOP)
```

Called with `$1` equal to `LOOP`, this macro allows to refresh the Robosoft on-board watch-dog.

Basically, this macro-call performs an empty read at the 0x0FF0000 MPC555 memory address. This action generate a chip select that asserts the watch-dog logic.

Here is an example of M4 macro-call used for refreshing the Robosoft on-board watch-dog:

```
> genWatchDog(LOOP)
```

20.1.3 The `END` option

```
genWatchDog(END)
```

Called with `$1` equal to `END`, this macro resets the Robosoft on-board watch-dog. At the current development stage, nothing special is done for hardware watch-dog finalization. This macro is not of interest.

20.2 The `watchDog_it_` user macro

```
watchDog_it_()
```

This user macro implements Robosoft on-board hardware watch-dog support. As for `inhAmp` macro, driving watch-dog is absolutly needed when you want to control motors. Calling `watchDog_it_` user macro requires no argument.

Depending on the SynDEx macro generation context (variable MGC), `watchDog_it_` macro call performs on-board watch-dog initialization, watch-dog refreshing (during main loop execution) and finally on-board watch-dog reset (during application finalization step). This macro is only expected to be under timer interrupt. Code generated for this macro is included into PIT handler. Thus watch-dog is refreshed asynchronously and periodically each time PIT trigs an interrupt. Please, when using this macro, check if your PIT period well fits the watch-dog refreshing constraints mentionned at subsection 20.1.1.

Here is a piece of M4 macro executive that handle the on-board watch-dog and periodically refresh it:

```
> main_
>   dnl Initialization step (MGC==INIT)
>   watchDog_it_()
>   loop_
>     dnl Loop execution (MGC==LOOP)
>     watchDog_it_()
>   endloop_
>   dnl Finalization step (MGC==END)
>   watchDog_it_()
> endmain_
```

Chapter 21

Dot matrix LCD controller support

Robosoft control board is equiped with a dot matrix LCD controller. This device can freely be used by programmers for any display operations. The following macros provide user with the most common Hitachi HD44780U display controller commands. Specific information about this device are found in the HD44780U data-sheet [6].

21.1 The `genLCD` generic macro

21.1.1 The `INIT` option

```
genLCD(INIT)
```

Called with `$1` equal to `INIT`, this macro performs the HD44780U LCD controller initialization. This macro-call, taking no more argument, is intended to be used for coding other convenient user oriented macros.

On-board HD44780U is controlled through QADC64 module A port A. To this aim, QADC64 module A port A is configured as a digital input/output interface. For, port A pins 6 to 0 direction is set as output, while port A pin 7 direction is set as input. With such an 8-bit interface, we operate LCD controller in 4-bit mode. HD44780U Datasheet [6] Figure 24 details the various steps involved in a HD44780U operating sequence. Table 21.1 gives the correspondence between HD44780U pins and QADC64 module A port A ones.

HD44780U pins	QADC64 module A port A pins
ENABLE	PQA6
RS	PQA5
$R/\overline{W}$	PQA4
DB7	PQA3
DB6	PQA2
DB5	PQA1
DB4	PQA0

Table 21.1: HD44780U and QADC64 module A port A pins assignments

Basically, initialization sequence, performs the following steps:

1. wait at least 40 ms after power on (i.e. after V_{cc} rises to 2.7 V)
2. send a first function set, with instruction byte 0x43

3. wait at least 4.1 ms (interface is 8 bits long)
4. send a second function set, with instruction byte 0x43
5. wait at least 100μs (interface is still 8 bits long)
6. send a third function set, with instruction byte 0x43 (interface is still 8 bits long)
7. send a fourth function set, with instruction byte 0x42 (interface is still 8 bits long)
8. now HD44780U controller enters the 4-bit mode interface
9. confirm we operate in 4-bit mode (setting DL to 0) and we use a 2 lines display (setting N to 1). Sent instruction byte is 0x2C. Refer to HD44780U Datasheet [6] pages 24 and 25 for a detailled description of available instructions.
10. set display on, cursor on and cursor blinking mode on. Sent instruction byte is 0x0F
11. set entry mode (increment address by one and shift cursor). Sent instruction byte is 0x06

After this, LCD controller is ready to receive display instructions. Instructions are sent in 4-bit mode, i.e. each instruction is sent in two time (refer to subsection 21.1.2 for further details about instruction writing).

21.1.2 The `WRINST` option

```
genLCD(WRINST,?inst)
```

(Please refer to documentation included in executive source code)

21.1.3 The `WRASCII` option

```
genLCD(WRASCII,ASCII_code)
```

(Please refer to documentation included in executive source code)

21.1.4 The `WRCHR` option

```
genLCD(WRCHR,?char)
```

(Please refer to documentation included in executive source code)

21.1.5 The `BUSYTST` option

```
genLCD(BUSYTST)
```

(Please refer to documentation included in executive source code)

21.1.6 The `RETHOM` option

```
genLCD(RETHOM)
```

(Please refer to documentation included in executive source code)

21.1.7 The CLRSCR option

```
genLCD(CLRSCR)
```

(Please refer to documentation included in executive source code)

21.1.8 The MOVFWD option

```
genLCD(MOVFWD)
```

(Please refer to documentation included in executive source code)

21.1.9 The MOVBCK option

```
genLCD(MOVBCK)
```

(Please refer to documentation included in executive source code)

21.1.10 The SETPOS option

```
genLCD(SETPOS,pos)
```

(Please refer to documentation included in executive source code)

21.1.11 The WRSTR option

```
genLCD(WRSTR,?char_buf)
```

(Please refer to documentation included in executive source code)

21.1.12 The END option

```
genLCD(END)
```

(Please refer to documentation included in executive source code)

21.2 The LCDdisp user macro

```
LCDdisp(?buf)
```

(Please refer to documentation included in executive source code)

Part IV

General purpose remarks

Chapter 22

About Robosoft board axis related signals

When using Robosoft board for driving axis, in conjunction with PWM signal, user may need to set both axis validation bit (i.e. axis active or inactive) and axis direction bit (i.e. backward or foreward). All of these operations are handled by `PWM` (c.f. 9), `inhAmp` (c.f. 19) and `dirAmp` (c.f. 18) user macros. These macros make use of MPWMSM and MPIOSM submodules, respectively for PWM and, validation and direction. Table 22.1 gives the correspondence between axis identifiers, MPWMSM submodule output pins and MPIOSM submodule data pins. Let us remark, that due to the actual Robosoft board hardware implementation, only the first four PWM signals are fully available (both duty-cycle and sign). This will propably change with future board revisions.

	PWM signal		Validation signal		Direction signal	
Axis Id.	PWM user channel	MPWMSM submodule output pins	Validation user signal	MPIOSM submodule data pins	Direction user signal	MPIOSM submodule data pins
0	0	MPWMSM0	0	D8	0	D4
1	1	MPWMSM1	1	D9	1	D5
2	2	MPWMSM2	2	D10	2	D6
3	3	MPWMSM3	3	D11	3	D7
4	4	MPWMSM16	(not available)			
5	5	MPWMSM17	(not available)			
6	6	MPWMSM18	(not available)			
7	7	MPWMSM19	(not available)			

Table 22.1: Correspondance between Robosoft board axis Id., PWM, validation and direction signals data pins

Last, let us note that validation user signals (i.e. MPIOSM pins from D8 to D11) can be controlled only if Robosoft software watchdog is activated (cf. chapter 20 for watchdog macros description).

Bibliography

[1] Thierry Grandpierre, Christophe Lavarenne, Yves Sorel.
SynDEx.
All information available at the SynDEx web site: http://www-rocq.inria.fr/syndex.

[2] Motorola Inc.
MPC555/556 User's Manual RISC PowerPC TM Microcontrollers
TSG Transportation Division.
Digital DNA From Motorola, Revised 15 October 2000.
All information available at the Motorola web site. The zipped document version can be downloaded at: http://www.maneapc.demon.co.uk/motorola/mpc555/555.zip. Document can also be read on-line at: http://www.maneapc.demon.co.uk/motorola/mpc555.htm.

[3] Motorola Inc.
Fast Quadrature Decode TPU Function (FQD)
(Order this document by TPUPN02/D).
Motorola Semiconductor Programming Note by Jeff Wright, 1997.
All information available at the Seattle Robotics Society web site. This document version can be downloaded at: http://www.seattlerobotics.org/encoder/200006/tpupn02.pdf.

[4] Analog Device
Quad 12-Bit Serial Voltage Output DAC DAC8420.
REV. 0.
All information available at the Analog Device web site. This document version can be downloaded at: http://www.analog.com/pdf/dac8420.pdf.

[5] Philips Logic
74HC/HCT4538 Dual retriggerable precision monostable multivibrator.
September 1993
All information available at the Philips Logic web site. This document version can be downloaded at: http://www.philipslogic.com/products/hc/pdf/74hc4538.pdf.

[6] Hitachi Semiconductor
HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver).
September 1999, Rev. 0.0
All information available at the Hitachi Semiconductor web site. This document version can be downloaded at: http://semiconductor.hitachi.com/products/pdf/99rtd006d2.pdf.

Index