USER GUIDE

1 ABOUT THIS GUIDE

Welcome to the ST9+ User Guide. The aim of this book is to help you to get a working knowledge of the ST9+ family. Using this foundation, you will be in a good position to understand and implement any of the ST9+ family microcontrollers. To make it easier, we have selected the major technical concepts of the ST9+ and will introduce them gradually over several chapters, always supporting theory with practical examples.

1.1 PREREQUISITES

This book addresses application developers. To fully benefit from the book content, you should be familiar with microcontrollers and their associated development tools.

For basic information on microcontrollers and development tools, you should refer to one of the many introductory books available on the subject.

1.2 RESULTS

The book will provide you with:

- A basic understanding of the ST9+ family
- Tutorial examples on installing and configuring ST9+ development tools
- Knowledge and ready-to-use examples on using ST9+ peripherals
- Reference applications describing a complete development flow
- Useful tips and warnings

Rev. 1.0

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1.3 HOW TO USE THIS GUIDE

As a first approach, we recommend you to study each chapter in sequence and carry out the exercises at each step.

1.4 COMPANION SOFTWARE

A downloadable file is available entitled ST9+ User Guide Companion Software, providing all the source text files, the listings, the object files and all other files mentioned in the text.

It also provides utilities for installing and configuring the tools and a handy text editor called the Programmer’s File Editor. This public domain text editor is supplied as a software package, in conformance with the author’s rights.

Naturally, if you have your own development environment, like Brief, Borland C++, etc. you can also use it.

You can download the ST9+ User Guide Companion Software from the http://www.st.com website product support page.

1.5 ABOUT THE AUTHORS

This User guide has been initially written by Jean-Luc Grégoriadès and Jean-Marc Delaplace and revised for the ST9+ by Jean-Luc Crébouw.

Jean-Luc Crébouw

A signal processing engineer, he has developed a voice synthesizer with a ST9+ and conducts ST9 trainings. He acts as a consultant field application engineer for all STMicroelectronics microcontrollers.

Jean-Marc Delaplace

A former electronics design engineer, he has worked throughout his career for various U.S. companies involved in lab automation equipment. He has used microprocessors since they first appeared on the market and programmed microcontrollers of various brands in industrial applications using both assembler and high-level languages.

Jean-Luc Grégoriadès

Teaches automated systems and industrial computer science at the Electrical Engineering department of the University of Cergy-Pontoise. He introduced the STMicroelectronics ST6 as a teaching base for his microcontroller course. On this occasion, he wrote with his friend J.M Delaplace, the book ”Le ST6 : Etude progressive d’un microcontrôleur” published at ”Editions DUNOD”.

1.6 RELATED DOCUMENTS

The following reference documents should be available for additional information:

– ST90158 - ST90135 Data Sheet

– ST9+ Programming Manual

– ST9 Family GNU Software tools V2.12

– ST9 GNU C Toolchain Release note V4.2

– ST9 Family GNU C Compiler

– GNU Make Utility

You can get a current list of documentation at http://www.st.com.

2 INTRODUCING THE ST9+ BASICS

The ST9+ microcontroller family has a common processor core surrounded by a range of powerful peripherals for interfacing many different devices. The peripherals have sufficient built-in intelligence to be able to perform even complex jobs on their own, freeing the core almost entirely from I/O handling. The core can thus be fully utilized for classical microprocessing tasks.

The ST9+ architecture is an original STMicroelectronics design, with the objective of providing an innovative and efficient microcontroller architecture dedicated to real-time control.

2.1 PROCESSOR CORE

When you compare different microcontrollers, you can estimate the relative computing power of the core, and also of the peripherals (if they include some intelligence). In some architectures, the peripherals make heavy use of the core and thus take up a part of its computing power. Many microcontrollers available on the market have a relatively powerful core, surrounded by very simple peripherals. This approach has the advantage of making the peripherals easy to use and configure but at the expense of the overall computing power.

The ST9+ is an example of a radically different compromise. Its core is among to the best 8-bit microprocessors on the market, in terms of computing performance and system management capabilities. It is assisted (rather than just surrounded) by peripheral blocks of which most can perform complex tasks without the intervention of the core. The net result is a powerful machine that can even perform impressive tasks just using its peripherals. The three applications described in this book give telling examples of processes handled solely by the peripherals.

The ST9+ is the latest generation of the powerful ST9 microprocessor family. The ST9+ core executes software more than three times faster than the ST9 core using optimized instructions and up to double the CPU frequency. The core as well as many peripherals have been enhanced, like for example, the Memory Management Unit (MMU), which is now more flexible, with a single 4-Mbyte memory space that is directly accessible without using bank switching. Another example is the Reset and Clock Control Unit (RRCU) which has features added for reducing power-consumption.

The ST9+ is a register-oriented machine. This means that a large number of registers is available in the core; but above all, this implies that the instruction set is tailored to make efficient use of its registers through optimized addressing modes. It is also well suited to the use of C language.

2.2 PERIPHERALS

The ST9+ family includes a large number of peripherals. The main ones are:

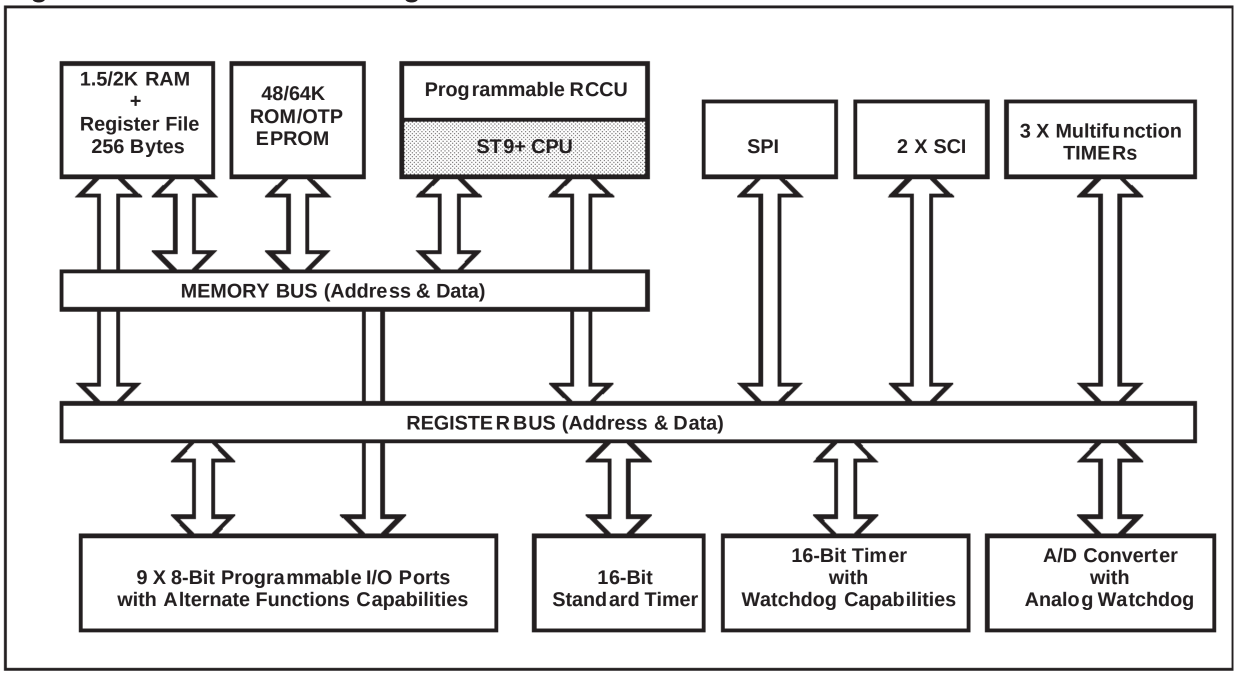
Acronym	Name	Function
MFT	Multi-Function Timer	All counting and timing functions. Includes auto-reload on condition, interrupt generation, DMA transfer, two inputs for frequency measurement or pulse counting, two outputs that can change on condition, PWM signal generation. Conditions include: overflow/underflow, comparison with one or two compare registers. Capture registers allow recording transitions on inputs with their time of occurrence.
SCI	Serial Communication Interface	Asynchronous transfer with either internal bit-rate generation or an external clock. Parity generation/detection. Address recognition feature that can request an interrupt on match of an input character. DMA transfer.
SPI	Serial Peripheral Interface	Serial input or output register, with internal or external clock. Intended for I/O expansion, or synchronous serial external device such as serial EEPROM (I2C and SPI protocol).
WDT	Watchdog timer	Can be used either as a watchdog or as a timer with input and output capable of pulse counting or waveform generation.
I/O port	Parallel input/output port	Parallel input/output ports. Each bit individually configurable as input, output, bi-directional, or alternate function. Inputs can be high impedance or with pull-up, CMOS or TTL-level. Outputs can have open drain or push-pull configuration.
ADC	Analog to digital converter.	Eight-bit analog to digital converter. One to eight channels can be converted in series. On each of two of the eight channels, an Analog Watchdog function allows two thresholds to be defined. When exceeded, an interrupt is generated.

These are the peripherals available on the standard variants. More peripherals are available on custom devices on request, e.g. a videotext decoding logic.

The ST9+ family includes so many variants it would go beyond the scope of this book to describe them all. They are all made up of the same ST9+ core surrounded by a set of peripherals and optionally ROM and/or RAM and/or EEPROM. This book has chosen to show the two most generic variants and provide a basis for understanding all the others.

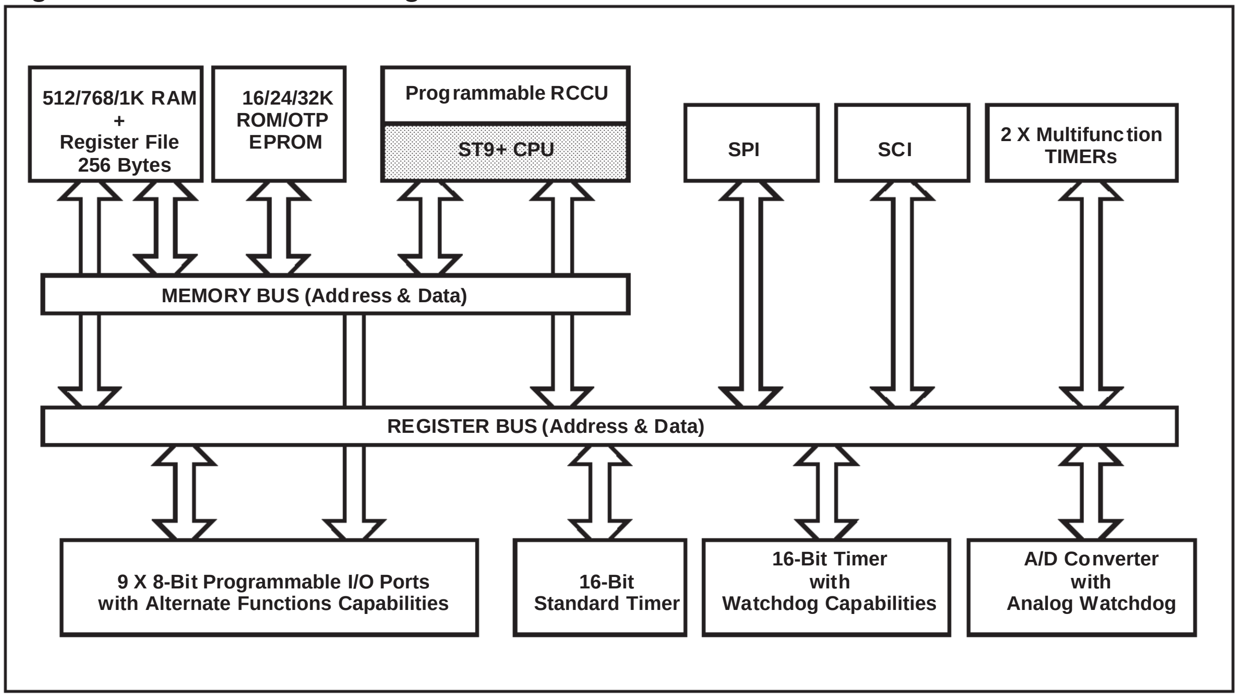
2.2.1 ST90158

Figure 1. ST90158 Block Diagram


1.5/2K RAM + Register File 256 Bytes
48/64K ROM/OTP EPROM
Programmable RCCU
ST9+ CPU
SPI
2 X SCI
3 X Multifunction TIMERs
MEMORY BUS (Address & Data)
REGISTER BUS (Address & Data)
9 X 8-Bit Programmable I/O Ports with Alternate Functions Capabilities
16-Bit Standard Timer
16-Bit Timer with Watchdog Capabilities
A/D Converter with Analog Watchdog

2.2.2 ST90135

Figure 2. ST90135 Block Diagram


512/768/1K RAM + Register File 256 Bytes
16/24/32K ROM/OTP EPROM
Programmable RCCU
ST9+ CPU
SPI
SCI
2 X Multifunction TIMERs
MEMORY BUS (Address & Data)
REGISTER BUS (Address & Data)
9 X 8-Bit Programmable I/O Ports with Alternate Functions Capabilities
16-Bit Standard Timer
16-Bit Timer with Watchdog Capabilities
A/D Converter with Analog Watchdog

3 PROCESSOR CORE: THE MAIN CONCEPTS

The term ST9+ designates a family of components. Each component shares the same core, surrounded by a particular configuration of memory and peripherals that make up the specific variant. The ST9+ core has a unique and powerful structure. This chapter explains the main building blocks that you need to get familiar with to be able to make full use of its capabilities.

The main features of the core architecture are:

– Register-Oriented Programming Model

– Single-Space memory addressing

– System and User Stacks

– Interrupt system with fully integrated controller

– Built-in DMA mechanism

– Reset and Clock Control Unit (RCCU) with PLL

3.1 ADDRESS SPACES

The ST9+ provides two different address spaces: Register Space and Memory Space. The Register Space draws its power from its size: 256 registers of which 224 are uncommitted, and from the fact that it can hold data or pointers to data that reside in any of the two spaces. The Memory spaces can address up to 4 Mbytes. This address space is arranged as 64 segments of 64 Kbytes to address Programs and as 256 segments of 16 Kbytes to address Data when the DMA is not used.

3.1.1 Register-Oriented Programming Model

The usual microprocessor core structure is based on an accumulator. The accumulator is the one register that holds the data to work on and the results of the arithmetic or logical operations applied to it. This structure has become a classic - for its simplicity - the internal data paths of the microprocessor all converge to the accumulator. The instruction set is simple, since you need to specify only one memory address in a data move instruction, the other being implicit: the accumulator itself.

This simplicity has its own drawbacks: the accumulator is the computation bottleneck, since to move data from one place in memory to another place, you have to do it through the accumulator. The simplest transfer involves at least two instructions: one to get the data, the other one to store it.

Register-Oriented models, in contrast, allow you to move data directly from one place to another in a single instruction. Data can come from a register or from a memory address and can go to either to a register or a memory address. You can code the addresses in the instruction, or store them in registers referenced by the instruction. This allows you to optimise your code by choosing to store frequently used data in registers, leaving less frequently used data in memory.

3.1.2 Register File

The ST9+ has a special addressing space for registers, providing 256 different register addresses. This large amount of registers gives you considerable flexibility in allocating variables. Register addresses are coded using one byte. You can use any of these registers to hold data or as a pointer either to other registers or to bytes in memory(1). Contrast this with processors that feature a certain number of registers, but in which some of these registers are meant to be only pointers or indexes, and some others not. These processors only allow transfers between memory and registers. The register organisation of the ST9+ gives you a real advantage you can make use of.

3.1.3 Direct access to the Register File

All the Register File can be accessed directly by their address prefixed by an ”R” except for register group D (13) that can only be addressed through the Working Register mechanism.

(1) Except for group E (14) reserved for the system registers and group F (15) reserved for the peripherals.

For example to address the register at the address 73, address it as R73, R49h or R1001001b in decimal, hexadecimal or binary. For a double register (16 bits) you can use the ”RR” prefix to address any data with an even address.

Any register can be given a user-defined name.

In C language:

```
register char data1  asm(”R73”);      /* = R73 */
register int  data2  asm(”RR60”);     /* = RR60 */
```

and accessed with:

```
   data1=13;
   data2=0x1234;
```

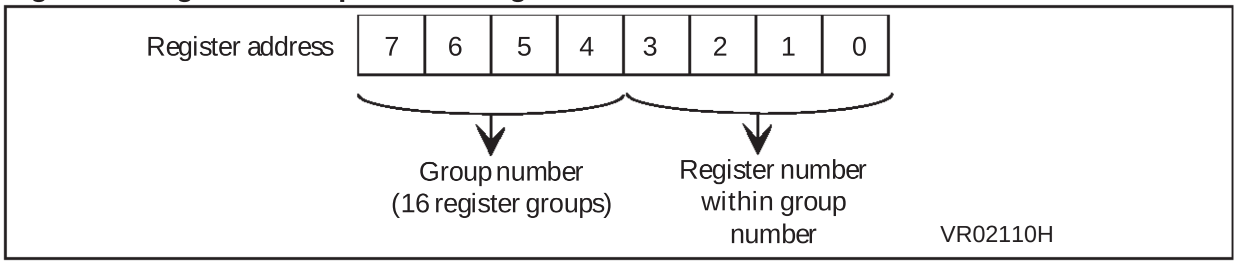
However using these registers needs an additional byte with the instruction mnemonic. Using Working Registers is more efficient, because it avoids using this address byte.

3.1.4 Working Registers

To further improve coding efficiency, a special mechanism has been created: the concept of working registers. This reduces to just 16 bytes the register space accessible by the instructions in the so-called working register addressing mode. Only four bits are required to address this space, allowing both the source and the destination of a data move to be coded in a single byte, thus saving both code size and execution time.

256 registers, when split into groups of 16, give 16 groups. The group used is indicated in a special register, called the **Working Register Pointer**. The register address is made up of the group number and the register address within the group, as follows:

Figure 3. Register Group Addressing Scheme


Register address
7
6
5
4
3
2
1
0
Group number (16 register groups)
Register number within group number
VR02110H

3.1.5 Peripheral Register Pages

All internal peripherals are mapped into the register space. Most of them have a multitude of features and can be configured in different ways. This implies that they have a large number of registers. Since only the last group of 16 registers is allocated for peripherals, a special scheme must be used to overcome this problem. It is called paging. The last group of registers actually addresses one pack of sixteen registers that belongs to the peripheral itself. Which pack of which peripheral depends on the value of a register called the **Page Pointer Register**.

There can be as many as 64 different pages, providing plenty of space for accessing peripherals.

Here are more details on these two mechanisms.

3.1.6 Working Registers and Register Pointers

The working registers offer a workspace of 16 bytes. This is sufficient for most applications, and much more convenient than a single accumulator. However, in some applications, this is still not enough. In this case you can easily allocate more than one register group to a particular program module. Since any register can be accessed directly, it is up to you to decide whether you want to switch working register groups or not to access the other groups of registers.

Since changing the current group involves only one instruction, the concept of working registers can greatly reduce context switching time, for example in the case of an interrupt service routine. Doing this preserves the contents of the whole group, and the reverse operation restores them, as in the example:

```
; the main program uses the working register group 0
InterruptRoutine:
   pushw   RPP         ; keep track of current group
   srp     #2          ; Switch to group 1 (see text below for details)
   ...
   ...                 ; Body of the interrupt service routine
   ...
   popw    RPP         ; Restore whatever group was active
   iret                ; Return from interrupt
```

Supposing we could not switch working registers, we would have to push 16 bytes to the stack to ensure that the contents of the working area have been preserved, and pop them back before returning. Obviously the example above is more efficient, both in code and data memory size, and also in execution time.

You use register pointers to allocate the working registers in a particular group. When writing in C or assembly language, you must position the working registers before you use them(2).

Switching groups involves the RPP register pair, made of the two registers RP0 and RP1. These registers are directly accessible, but since their bits are laid out in a nontrivial manner, it is recommended that you set them using only one of the `srp`, `srp0` or `srp1` instructions.

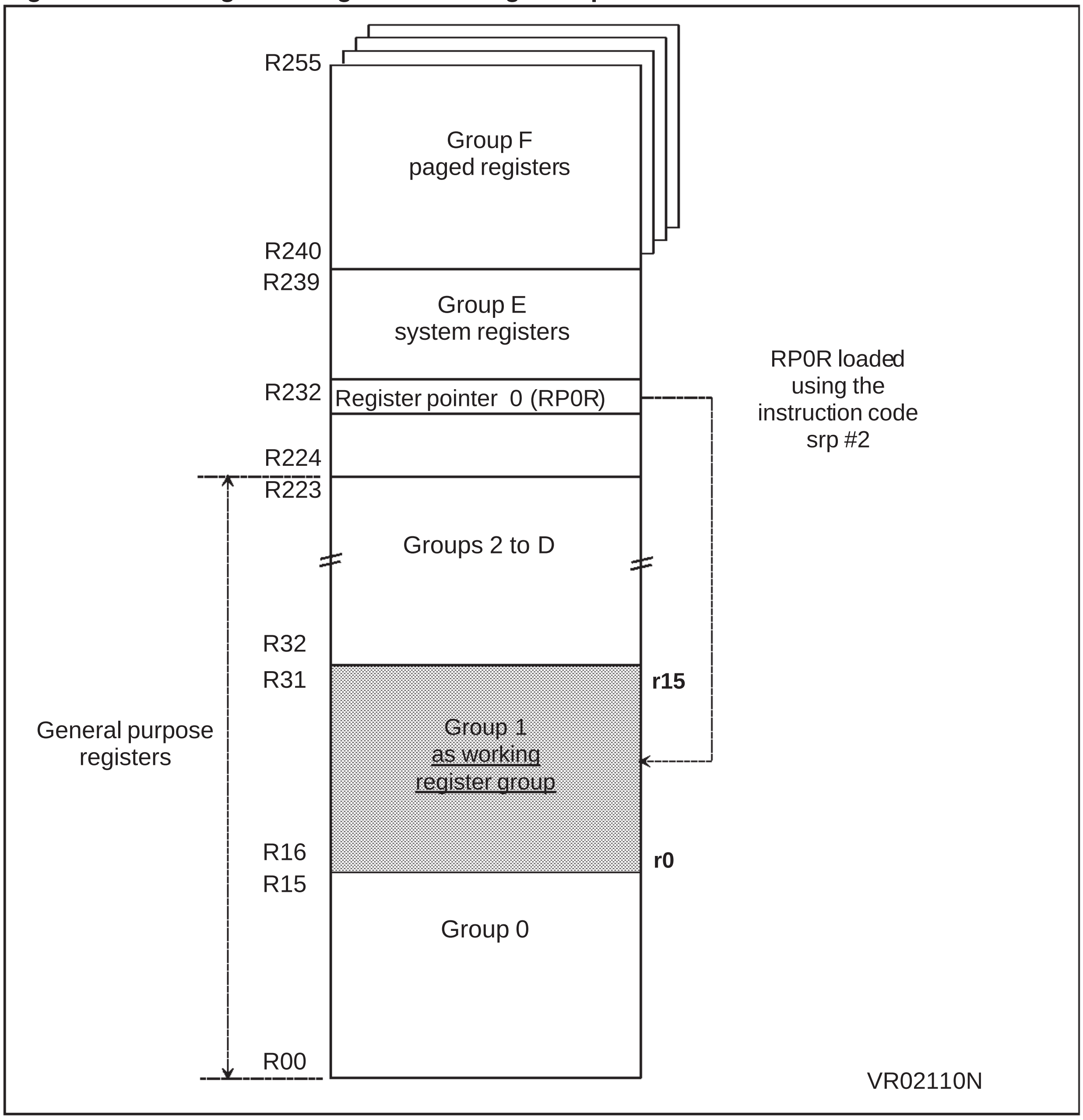
(2) ST90158 - ST90135 Data Sheet; Address spaces of the register file § 2.2.1 and following, system registers § 2.3

The registers are considered as sixteen groups of sixteen registers each. This is the way they are represented in the register number summary table (See Table 1.). Use the numbers in this table to refer to a register directly, e.g. when writing R35, this designates the fourth register of group 2.

3.1.6.1 Switching the 16 Groups of Working Registers

This is done using the `srp` instruction. In spite of what we have explained up till now, and how it is usually represented, the core does not actually divide the registers in 16 groups of 16 registers, but 32 blocks of 8 registers. This is why the `srp` instructions require arguments ranging from 0 to 31 instead of 0 to 15. Here is how the Register Pointers select the desired register group among 16 such groups.

Figure 4. Selecting a 16-Register Working Group



Using `srp` defines one group of sixteen working registers named r0 to r15, and occupying 16 contiguous registers in the register file. The lowercase r for the register number indicates that it is a working register number, in contrast to uppercase R registers that indicate an absolute register number. For example, accessing r3 is the same as accessing R19 if the current group is group 1:

```
srp #2   ; Switch to group 1
inc r3   ; increment 4th register of the group
```

```
   inc R19  ; increment the same register again
```

The following table summarises the use of the `srp` instruction and its effect in terms of group selection.

Table 1. Register Page Number Summary.

Hexadecimal register number	Decimal register number	Function	Register group decimal (hexadecimal)	srp #n instruction to select a group to provide r0-r15
F0-FF	240-255	Paged registers	15 (F)	srp #30
E0-EF	224-239	System registers	14 (E)	srp #28
D0-DF	208-223	General purpose registers	13 (D)	srp #26
C0-CF	192-207		12 (C)	srp #24
B0-BF	176-191		11 (B)	srp #22
A0-AF	160-175		10 (A)	srp #20
90-9F	144-159		9	srp #18
80-8F	128-143		8	srp #16
70-7F	112-127		7	srp #14
60-6F	96-111		6	srp #12
50-5F	80-95		5	srp #10
40-4F	64-79		4	srp #8
30-3F	48-63		3	srp #6
20-2F	32-47		2	srp #4
10-1F	16-31		1	srp #2
00-0F	00-15		0	srp #0

Notes: Though it is possible, it normally makes no sense to set the working register group either to group E (14) or F (15), since the registers in these groups have predefined meanings. You cannot use them to store intermediate values of calculations without greatly affecting the behaviour of the microcontroller in an unpredictable way. However, bit-level instructions are only available using working register addressing, so when you need to do bit manipulations in these groups, setting the register pointer to either 28 or 30 is an efficient way of programming when accessing these two groups.

The `srp` instruction is the only one you have to use to switch register groups, and is the way working registers are used in most programs. However, the working register scheme includes a subtlety that is seldom used, but that could give you even more flexibility in some cases. This is what is described in the next paragraph.

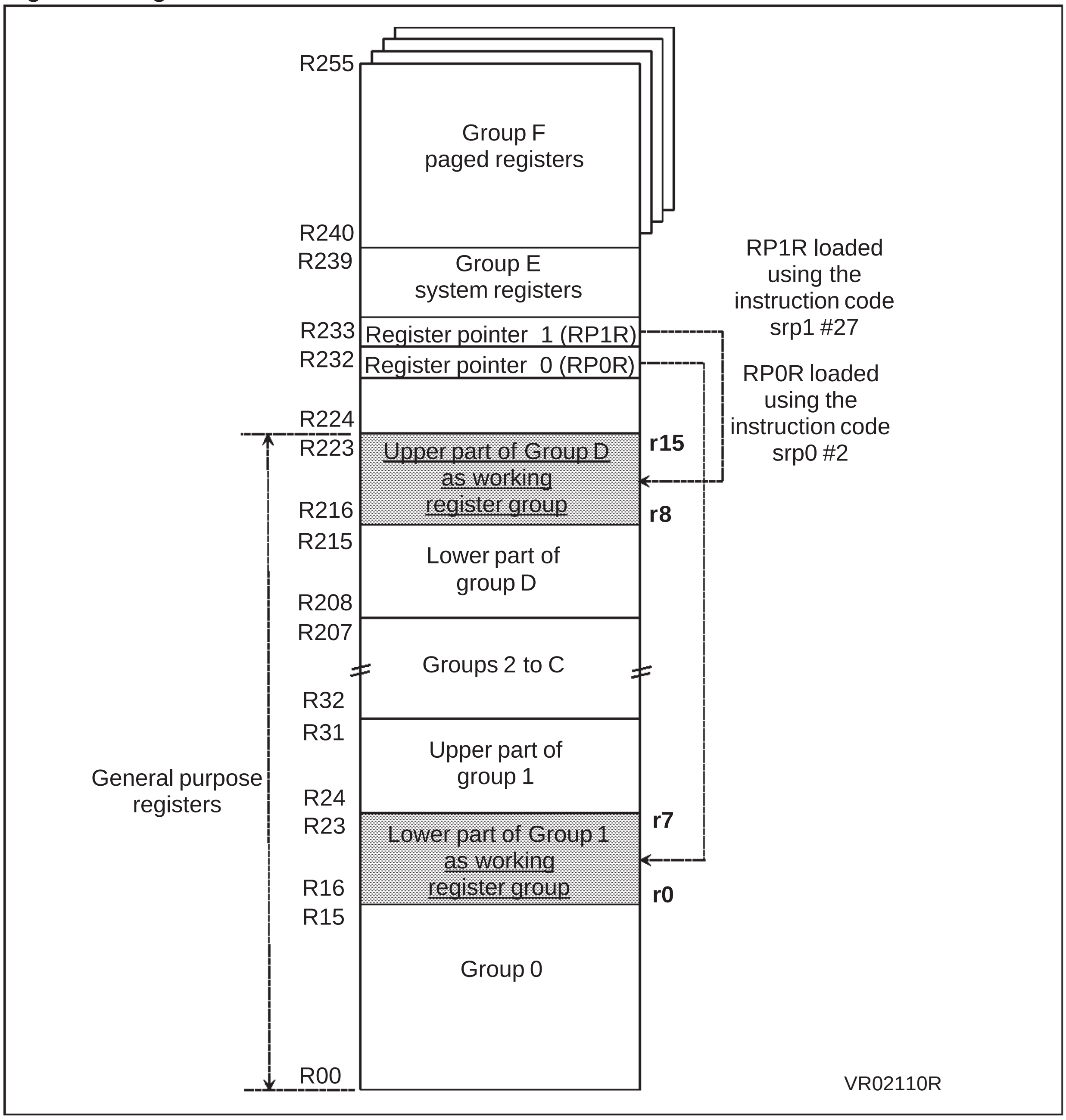
The same register group number can be selected by odd or even number. In fact the formula is:

```
   srp     2*n+1      ; for group n
or
   srp     2*n        ; for group n
```

3.1.6.2 Defining Two Separate Groups of Eight Working Registers

In this mode, the `srp` instruction is not used. Instead, we use the pair of instructions `srp0` and `srp1`. When using a working register, r0 to r7 address the first to the eighth register of the whole group selected by half the value in RP0 i.e. all the registers of the half group selected by RP0. Registers r8 to r15 relate to the first to the eighth register of the group pointed to by RP1. Here is how the two blocks are selected:

Figure 5. Register Numbers



As an example, if RP0 is set to the half group 2 (lower part of whole group 1) and RP1 to the half group 27 (upper part of whole group 13), r0 will designate R16 (2 x 8 + 0) while r15 designates R223 (27 x 8 + 7).

Using either method depends on the organisation of the data in the register file. You may find it convenient to use two eight-register blocks if you need to make quick calculations on pairs of data that are far apart in the register file.

The page numbering and switching instructions are summarised below:

Table 2. Register Page Number Summary

Hexa-decimal register number	Decimal register number	Function	Eight-Register block decimal (hexadecimal)	srp0 #n (or srp1) #n instructions to select a block to provide r0-r7 and r8-r15 respectively
F8-FF	248-255	Paged registers	31 (1F)	srp0 or srp1 #31
F0-F7	240-247	Paged registers	30 (1E)	srp0 or srp1 #30
E8-EF	232-239	System registers	29 (1D)	srp0 or srp1 #29
E0-E7	224-231	System registers	28 (1C)	srp0 or srp1 #28
D8-DF	216-223	General purpose registers	27 (1B)	srp0 or srp1 #27
D0-D7	208-215		26 (1A)	srp0 or srp1 #26
C8-CF	200-207		25 (19)	srp0 or srp1 #25
...	...		...	...
78-7F	120-127		15 (0F)	srp0 or srp1 #15
...	...		...	...
20-27	32-39		4	srp0 or srp1 #4
18-1F	24-31		3	srp0 or srp1 #3
10-17	16-23		2	srp0 or srp1 #2
08-0F	08-15		1	srp0 or srp1 #1
00-07	00-07		0	srp0 or srp1 #0

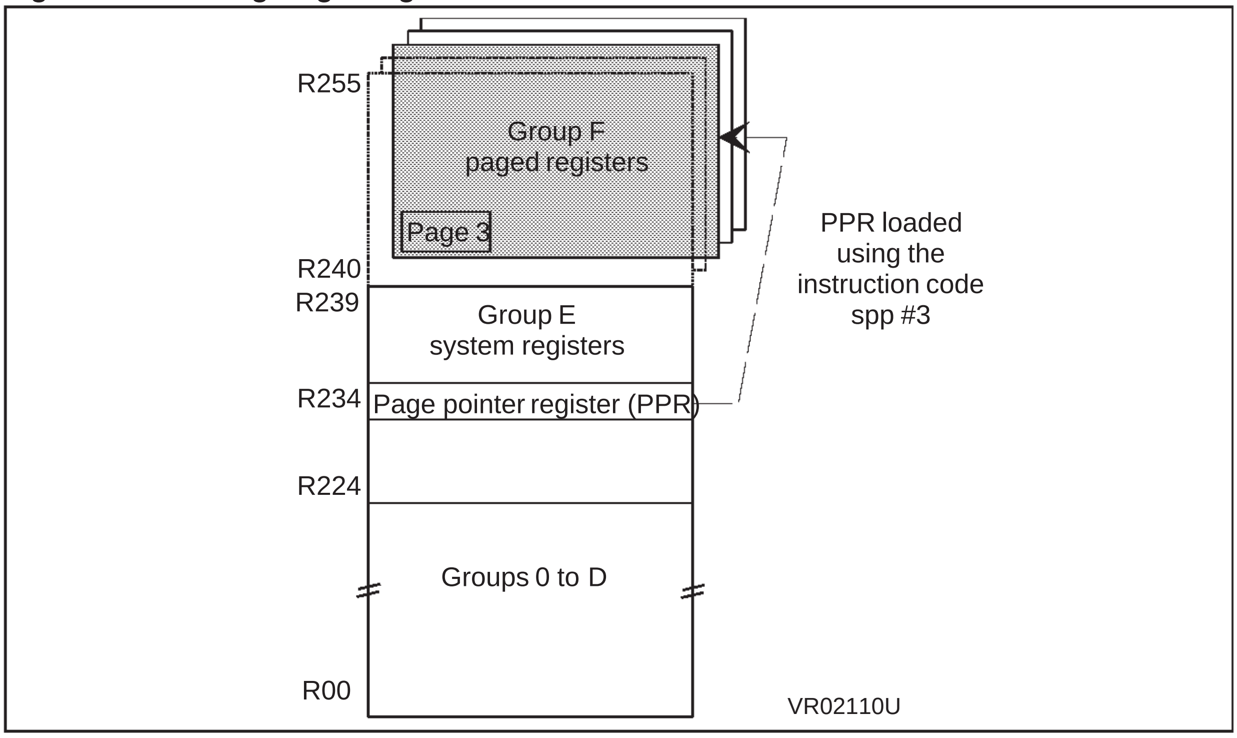
When you use eight-register or 16-register groups, you may very likely have a subroutine or an interrupt routine that uses a different set of working registers. You must save (push) the pair of register pointers RPP that include RP0 and RP1 at the beginning of the routine and restore them on exit.

3.1.6.3 Peripheral Register Paging

Group F of the sixteen register groups is paged so that as many as 64 different groups can be mapped to this address range. This large space is used to accommodate the registers related to the peripherals. The paging technique, allows you to add any number of peripherals and still be able to handle them without using up more addresses in the register space.

When you access a register in group 15, first set the Page Pointer Register to the number of the page that contains the register you want. Here is how a page is selected:

Figure 6. Selecting Page Registers



As with the working registers, if a subroutine or an interrupt routine needs to access a peripheral that uses paged registers (which is very likely), you must save (or push) the register pointer PPR at the beginning of the routine and restore it on exit.

Notes: In both assembly and C languages, include files are supplied with symbolic names predefined for all the peripherals. These names are unique for each peripheral; however several different names relate to the same register, but in a different page. You must bear in mind that writing for example (in C language):

```
S_ISR = 0 ;  /* clear serial peripheral error register  */
```

does not automatically select the proper page; this statement must be preceded by another one that selects the SCI page. Since no predefined C statement exists for this, a convenient way is to define an assembler statement under the form of a macro that will read nicely in the C source.

An example of the correct way to access the SCI register is:

```
#define SelectPage(Page) asm (”spp %0”:: ”i” (Page)) ; /* pseudo function
   to select a page */
   SelectPage( SCI1_PG ) ;/* select the serial peripheral page */
   S_ISR = 0 ;                /* clear serial peripheral error register */
```

3.1.7 Memory Management Unit

Like most microcontrollers, the ST9+ has a bus for interfacing internal and external memories. This allows you to store both programs and data. A special feature of the ST9+ is that it can address a 4 Mbyte single space to address ROM, RAM, EPROM, EEPROM, FLASH.

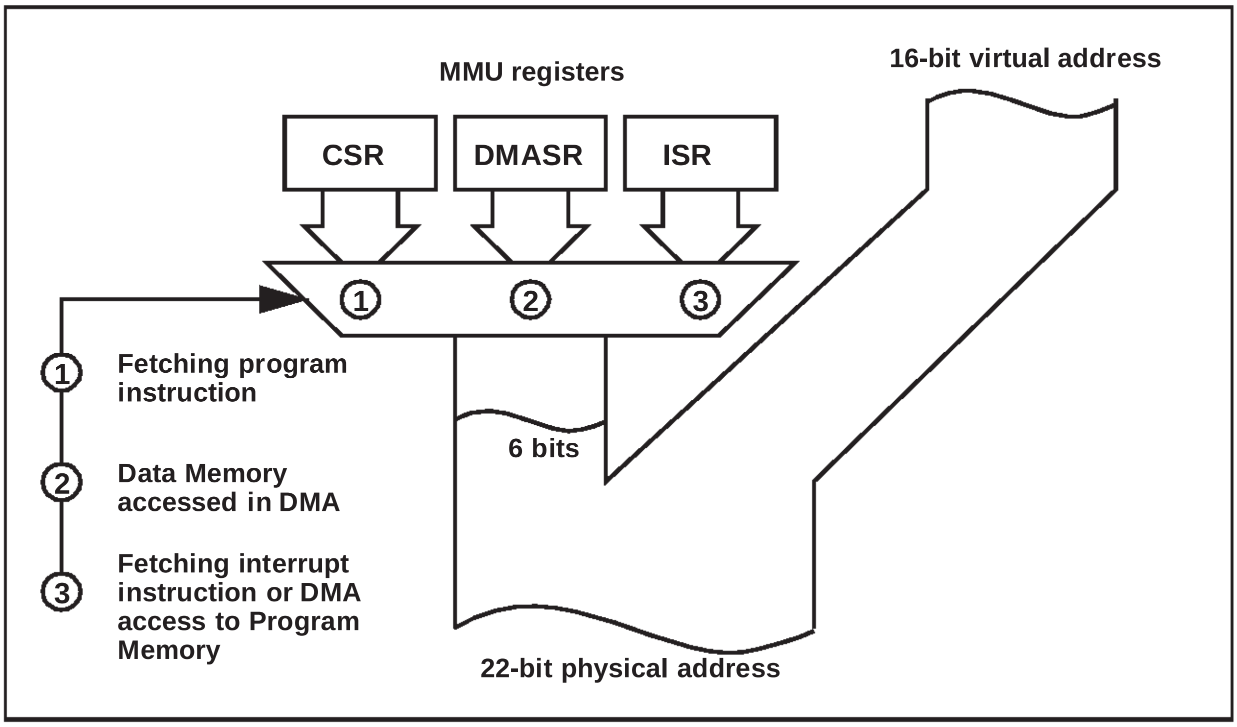
To address the 4 Mbytes of memory, the address bus is 22 bits wide. To manage the 22-bit address with 16-bit direct or indirect addressing, the memory mechanism adds extra bits to the 16-bit address and then works with segments (see Figure 7) or pages (see Figure 10). The memories are arranged in 64 segments of 64 Kbytes for the program and in 256 segments of 16 Kbytes for data.

A set of special registers are used to extend the 16-bit address. Programs use the CSR, DMA uses the DMASR or ISR and interrupts use the ISR or CSR register to provide the 6 Most Significant Bits to make a 22-bit address. Data uses a set of four registers (DPR0-3) to provide the 8 Most Significant Bits to make a 14-bit address.

Data can be addressed in the Program segment by using special move instructions: `lddp`, `ldpp`, `ldpd`, `lddd`.

It is easier from a hardware point of view to use only one address space for Program and Data.

Figure 7. Addressing via CSR, ISR and DMASR


MMU registers
16-bit virtual address
CSR
DMASR
ISR
1
2
3
1 Fetching program instruction
2 Data Memory accessed in DMA
3 Fetching interrupt instruction or DMA access to Program Memory
6 bits
22-bit physical address

3.1.7.1 Program Segment

We can consider this memory as linear since we can jump anywhere in memory space using the special JPS, CALLS and RETS instructions.

The 6-bit CSR register is used to extend the 16-bit address to a 22-bit address by concatenation (see Figure 7). To make a fast branch in the same segment, use the common JP, CALL and RET instructions.

To branch to another segment using a far call, the use of CALLS saves the current PC value and the CSR value (Code Segment Register) in the stack, before loading the PC and the CSR registers with the new values. Every time the segment changes you have to use the far branch even if you branch from address (n)FFFFh to (n+1)0000h. This is because the program doesn’t manage the 6 high-order bits of the 22-bit program addresses if you don’t use a far branch to change the CSR register value. The script file (described in the Development Tools chapter) allows you to place all your program modules anywhere in a single segment.

The far branch instruction adds only 2 to 4 additional cycles compared to a near branch instruction.

Note: In C language, using the small code model (this means only one 64 Kbyte segment is used), all calls use local branches. If more than one segment is used, the large code model is required and all calls use far branches even when branching locally. To avoid far calls in the same segment, the segment to be called has to be declared as **static** if it is not called from another segment.

3.1.7.1.1 Segment and Offset in assembler mode or C language

The Offset represents the address in the segment with 16 bits. If the Segment and the Offset are known the syntax of far branches are:

```
jps     segment,offset       ; 6 bits + 16 bits
jps     symbol               ; 22 bits
calls   segment,offset       ; 6 bits + 16 bits
calls   symbol               ; 22 bits
calls   (R),(rr)             ; 6 bits + 16 bits
calls   (r),(rr)             ; 6 bits + 16 bits
rets                         ; 22 bits
```

The assembly tools accept a set of directives which retrieves the elements of a function address or of a label.

The operator **SEG** (stands for SEGment) allows you to extract the segment number of a label; similarly, the operator **SOF** (stands for Segment OFfset) allows you to extract the offset of a label within its segment.

These operators are especially useful when applied to a function or instruction label, although the macro-assembler and assembler do not verify the type of the label.

Example:

```
ld      r0,#SEG Function      ; extract the 6-bit segment
ldw     rr2,#SOF Function     ; extract the 16-bit offset
calls   (r0),(rr2)
```

The same functions exist in C language: SEG(Function); SOF(Function).

3.1.7.2 Interrupt Service Routine Segment

One program segment is reserved for storing the Interrupt Service Routines. All the interrupt routines start in this segment.

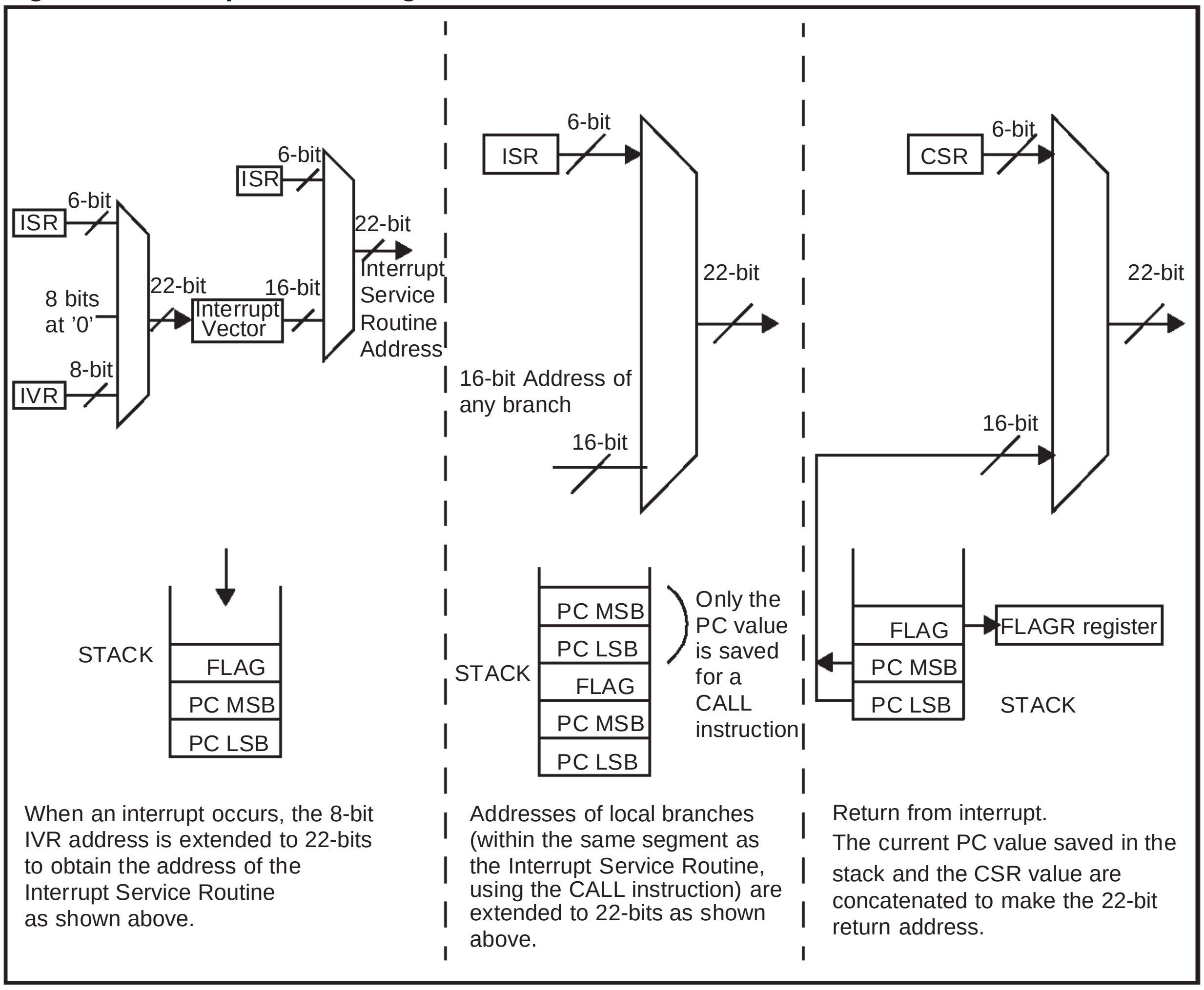
To obtain a 22-bit address, the 16-bit address is concatenated with 6 bits from the ISR register (the 6 bits from the ISR register are the high-order bits of the address).

To offer compatibility with the previous ST9 versions and to have a new powerful address mechanism, you can select ”ST9old” or ”ST9+” mode using the EMR2 bit in the ENCSR register.

Both modes use the concatenation of the ISR register value and the 16-bit address as shown in Figure 7 to address the interrupt vector.

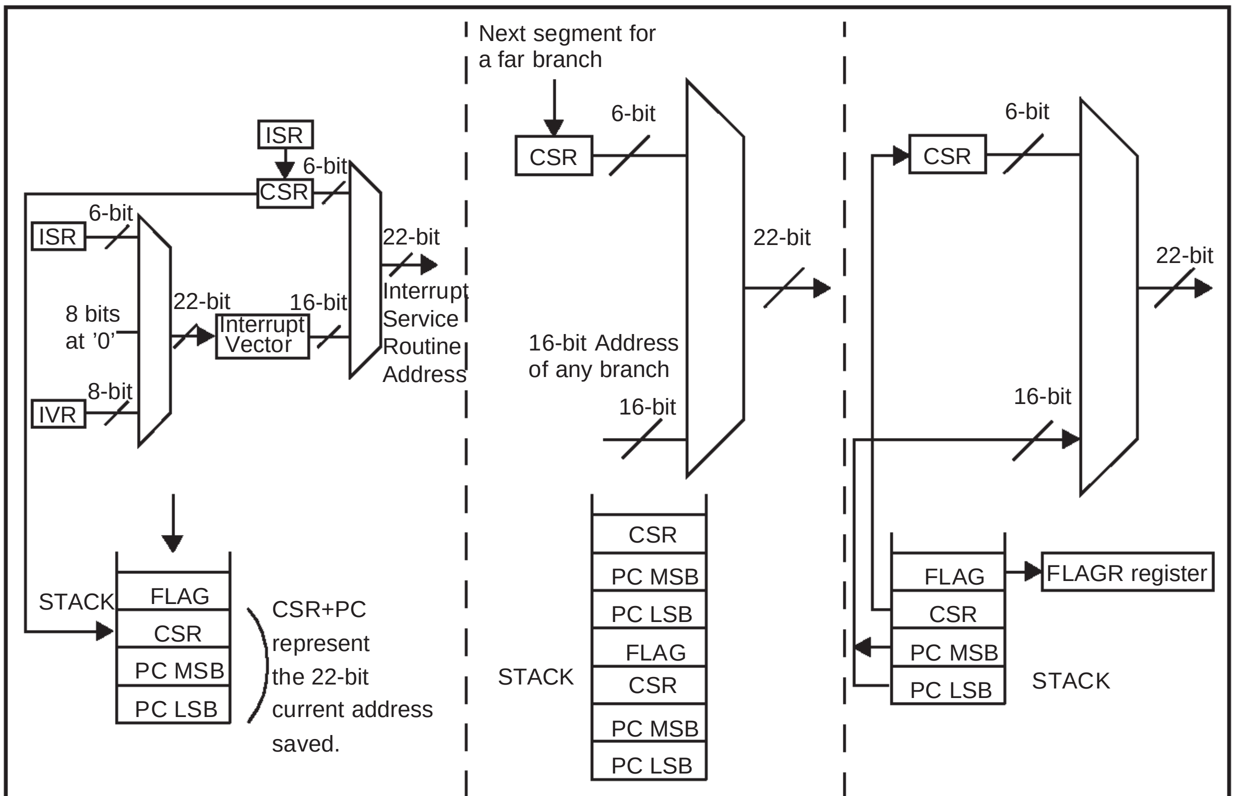
Then, in ”ST9old” mode, only the ISR register value is used during the interrupt routine. So it’s not possible to jump to another segment from the Interrupt Service Routine because the CSR register value is not saved with the FLAGR value and the current PC value when the interrupt occurs. The advantage of ”ST9old” mode is to reduce stack memory usage and CPU cycles by not saving the CSR value (see Figure 8). This figure shows you the different mechanisms that are used when an interrupt occurs, when a branch or a call is executed during the interrupt routine and when the return from interrupt instruction (RETI) is executed.

Figure 8. Interrupt Processing in "ST9old" Mode


ISR
6-bit
8 bits at '0'
8-bit
IVR
22-bit
Interrupt Vector
16-bit
ISR
6-bit
22-bit
Interrupt Service Routine Address
STACK
FLAG
PC MSB
PC LSB
When an interrupt occurs, the 8-bit IVR address is extended to 22-bits to obtain the address of the Interrupt Service Routine as shown above.
ISR
6-bit
22-bit
16-bit Address of any branch
16-bit
STACK
PC MSB
PC LSB
FLAG
PC MSB
PC LSB
Only the PC value is saved for a CALL instruction
Addresses of local branches (within the same segment as the Interrupt Service Routine, using the CALL instruction) are extended to 22-bits as shown above.
CSR
6-bit
22-bit
16-bit
FLAG
PC MSB
PC LSB
FLAGR register
STACK
Return from interrupt.
The current PC value saved in the stack and the CSR value are concatenated to make the 22-bit return address.

In "ST9+" mode, saving the CSR value allows you to change its value to branch to another segment. When the interrupt occurs and when the current PC value, the CSR value and the FLAGR value are saved, the ISR register value is stored in the CSR register (see Figure 9).

Figure 9. Interrupt Processing in ”ST9+” Mode



When an interrupt occurs, the 16-bit Interrupt Vector is extended to 22-bits as shown above.

Before loading the CSR register with the ISR value, the CSR value is saved in the stack.

Addresses of local and far branches (to any segment) are extended to 22-bits as shown above.
For a far CALL, the CSR value is saved with the current PC value in the stack and the CSR register is loaded with the new segment to jump to.

Return from interrupt.
The current PC value and the CSR value saved in the stack are concatenated to make the 22-bit return address.

3.1.7.3 DMA segment

To address the total 4Mbytes of memory in a DMA transaction, the DMASR points to a 64 Kbyte segment. Since there is no need to have more than one segment at the same time for the transaction, the DMA uses a single 64Kbyte segment instead of 4 pages like a Data Segment (see below). To use the DMASR register, the DP bit in the DMA Address Register (DAPR) must be set. If DP is reset, the DMA uses the ISR register instead of the DMASR register.

3.1.7.4 Data Segment

Data uses the page mechanism to address the 4 Mbytes of memory.

To authorize data coming from different 64 Kbyte segments, a set of 4 Data Page Registers (DPR0 to DPR3) allows you to address 16 Kbytes per register (see Figure 10). The DPR is selected with the 2 high-order bits of the 16-bit data address:

DPR0: from 0000h to 3FFFh (b15-b14=00)

DPR1: from 4000h to 7FFFh (b15-b14=01)

DPR2: from 8000h to BFFFh (b15-b14=10)

DPR3: from C000h to FFFFh (b15-b14=11)

After you select the DPR, the 8-bit value of the selected DPR register is used to extend the 14 remaining bits of the address to 22 bits.

For example if DPR0 equals 20h and DPR1 equals 2h, each memory access in the range of 0000h to 3FFFh uses the DPR0 page and addresses data from 080000h to 083FFFh, and each memory access in the range of 4000h to 7FFFh uses the DPR1 page and addresses data from 008000h to 00BFFFh (see Figure 10). For example:

16-bit address = 0010 0101 1010 0101 = 25A5h   0000h < 25A5h < 3FFFh

DPR0 is selected, so the 6 high-order bits are equal to 20h

22-bit address = $\underline{\text{0010 0000}}$ $\underline{\text{10 0101 1010 0101}}$

DPR0 value, 14 LSB of the 16-bit address

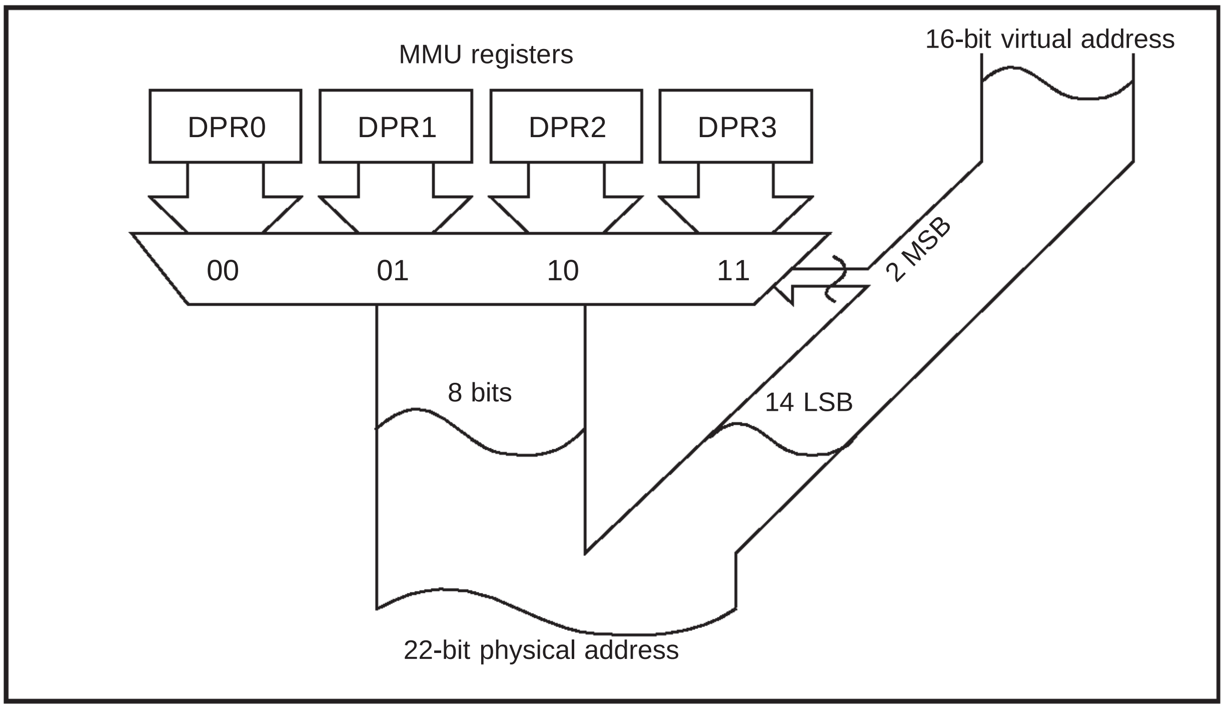
= 00 1000 0010 0101 1010 0101 = 0825A5h

With this mechanism, if the 16 bits addresses are different only on the 2 highest bits and if all the DPR registers selected with these two bits have the same value, the resulting 22-bit address will be the same.

Four pages of 16Kbyte of data memory are enough for many applications and allow you to use data from different segments without costing additional CPU cycles. With four DPRs, you can access up to 64K (4 x 16K) of data without changing the DPR values. Data can be variables stored in RAM or constants stored in program ROM.

The four DPR registers are located in the MMU register page (page 21 of register group F). If you use them frequently, you can relocate them to register group E, by programming bit 5 of the EMR2 register (R246 in page 21). This avoids you having to switch to the MMU register page from another peripheral register page in order to change a DPR register value.

Figure 10. Addressing via DPR0-3



3.1.7.4.1 Accessing the Page and the Offset in assembler or C language

The assembly tools implement a set of operands which allows you to extract the components of a data address.

The **PAG** operator (stands for PAGe) extracts the page number of an address; similarly, the **POF** operator (stands for Page OFfset) extracts the offset of the address within the page.

Syntax:

PAG label

POF label

Be careful that directly using the data label and using the POF operator on a data label are not equivalent: the data label gives the 16 bits of the logical label address; the POF operand gives the 14 lowest bits of the label's physical or logical address.

Example:

Assuming data is mapped at address 0x129876:

```
ldw     rr2,#POF data    ; rr2 = 0x1876
ldw     rr4,#data        ; rr4 = 0xD876 (with DPR3=0x4A)
```

Remember that you must take care of which data pointer has to be set before accessing a variable.

Example:

Assuming that data has been mapped in a page aligned on address 0xC000, this means that DPR3 will be used, therefore the following code is correct:

```
ld      DPR3,#PAG data
ldw     rr2,data
ld      r4,(rr2)
```

In assembly language, it is possible to access data through another DPR:

Example:

Still using data at an address aligned with 0xC000, following code is correct:

```
ld      DPR2,#PAG data          ; if data address=0x01C765, DPR2=7
ldw     rr2, #(POF data)+0x8000 ; rr2=0x8765
                                ; #(POF data) to reset bit 15 and 14
                                ; and 0x8000 to use DPR2
ld      r4,(rr2)                ; indirect addressing mode
...
ld      r4, (POF data)+0x8000   ; direct addressing mode
```

Note: Look at the explicit usage of the immediate addressing mode (#) to get the page number and the offset; it is consistent with the ST9 assembly syntax shown in the following example:

```
ldw     rr2,#Var
ld      r4,(rr2)
...
ld      r4, Var
```

The same functions exist in C language: PAG(data); POF(data);

3.1.7.5 ISR and CSR program example

We can use an example to show the use of the CSR and ISR registers.

Look at Figure 12 in the Interrupt Vector chapter and Figure 9 to help you to follow the PC value when interrupt occurs. Before looking at the interrupt management we have to initialize the program.

This example is close to the ST92R195 examples given in the Companion software in directory gnu9p\samples\st92r195\.

The purpose is to increment a variable in a subroutine and to reset it with an interrupt coming from the WDT counter (Watchdog Timer) end of count.

The Interrupt Service Routine Vector must contain the Interrupt Segment Register address which is the name of the C function ”ISR_WDT”. The Vector must be at the beginning of the first 256 bytes of the Interrupt Segment.

```
;**************** Interrupt Service Routine Vector *****************
IntVect = 10h; origin of interrupt vector table
   .extern ISR_WDT
   .global IntVect
; Reset and interrupt vectors.
   .text
   .org    IntVect
   .word   ISR_WDT          ; INTA0 interrupt vector
   .blkb   50               ; reserve room for the interrupt vectors
```

Here is the FILE0.C containing the main program called by the CRT9F.ASM start-up program. This file is in the segment 0.

```
#include ”system.h”
#include ”mmu.h”
/* declare prototypes of external functions used in main */
extern void Segment1_Func (void);
extern void ISR_WDT (void);
extern void InitWDT (void);
#define SelectPage( Page ) asm ( ”spp %0”:: ”i” (Page) ) ; /* select page */
/* Variable */
int counter;
void Initialization (void)
{
   SelectPage( MMU_PG ) ;
   ISR = SEG(ISR_WDT);   /* initialize the ISR to ISR_WDT segment */
   EMR2|= EMR2_encsr;  /* works as ST9+ by setting to 1 the Enable Code
                          Segment Register*/
}
/* main function, launched by the startup program CRT9F.ASM */
void main (void)
{
 counter = 0;
/*Call Initialization in same segment, this will nevertheless generate a
```

```
   calls because Initialization() is not declared as STATIC*/
 Initialization ();
 InitWDT ();
 while (1){
   Segment1_Func ();
 }
}
```

The initialization program will load the ISR register with the Watchdog Interrupt Subroutine segment by giving the name of the program which is ”ISR_WDT()”.

To use the segment mechanism for interrupts, the ST9+ has to be initialized as ”ST9+” instead of ”ST9old” by setting the EMR2 bit in the Enable Code Segment Register (ENCSR).

We call the InitWDT function which is in the segment 0 and initializes the WDT.

```
   .include ”system.inc”  ; System register
   .include ”page_0.inc”  ; Page 0 registers
   .global InitWDT
SHORT = 1000;
; Reset and interrupt vectors.
   .text
InitWDT:
   spp     #WDT_PG                ; select watchdog page
   ld      WDTPR,#127             ; prescaler 1/128
   ldw     WDTR,#SHORT            ; Short time
   ld      WDTCR,#WDm_stsp        ; continuous mode, start operation
   spp     #EXINT_PG              ; external interrupt page 0
   ld      EIVR,#EIm_tlism+IntVect
   ; TLIS = 1, service routine vectors start at 10h
   ld      EIMR,#EIm_ia0m         ; maskable int. : INTA0 enabled
                                  ; all others disabled
   ld      EIPLR,#00000001B       ; priority levels
   ld      CICR,#Im_ienm+7        ; Interrupts enabled, program level = 7
   rets
```

Now in the infinite loop we call the function ”`Segment1_Func` ()” which is in the segment 1 and increments the variable counter.

```
#include ”mmu.h”
#include ”stdio.h”
```

```
#include ”ctype.h”
extern int counter;/* address of counter = 0x2000 in DPR0 page */
int init_counter=1234;/* address of init_counter = 0x8000 in DPR2 page */
static void Segment1_Func1(){
   if((counter&3)==3) counter++;
   if(counter==128) counter=init_counter;
}
void Segment1_Func()
{
   counter++;
   Segment1_Func1();
}
```

The call to Segment1_Func1() is local because of the static declaration. CALL and RET are used instead of CALLS and RETS.

The CSR, which is equal to 0 before it was called, will be loaded with the value ’1’ and the previous CSR will be stored in the system stack with the next PC address for the return.

The stack is stored with:

Stack	data	register saved	meaning
00	00 081d	CSR value	segment 0 - call from main (CSR+PC)
08		PC MSB	
1d		PC LSB	
00	00 009e	CSR value	segment 0 - far call from CRT9F routine (CSR+PC)
00		PC MSB	
9e		PC LSB	

When the WDT end of count is reached, the program is interrupted. The values of PC, CSR and FLAGR registers are saved in the stack before changing the segment by loading the ISR value in the CSR register.

If an interrupt occurs during the execution of the ”Segment1_Func()” function in segment 1 the stack could be loaded like this:

Stack	data	register saved	meaning
xx		r6	
xx	xxxx	r5	rr4
xx		r4	
xx	xxxx	r3	rr2
xx		r2	
xx	xxxx	r1	rr0 - in the example the Interrupt Service Routine is a C program which save the 3 registers rr0, rr2 and rr4

xx		r0	
01	01		FLAGR value
01	01 0000	CSR value	segment 1 - interrupt in `Segment1_Func` () (CSR+PC)
00		PC MSB	
00		PC LSB	
00	00 081d	CSR value	segment 0 - call from main (CSR+PC)
08		PC MSB	
1d		PC LSB	
00	00 009e	CSR value	segment 0 - call from CRT9F (CSR+PC)
00		PC MSB	
9e		PC LSB	

Here is the Interrupt Service Routine which resets the variable counter.

```
#include ”mmu.h”
#pragma interrupt (ISR_WDT)
int extern counter;
void ISR_WDT ()
{
   counter = 0;
}
```

You can find an example script file in the gnu9p\samples\st92r195\ directory with the ”segment.scr” file (see Section 5 on Development Tools for details).

3.2 STACK MODES

The ST9+ allows you to have two separate stacks: a system stack and a user stack. The core uses the system stack in interrupt routines and subroutines to save return addresses, the flag register and the CSR depending on option (EMR2 register bit Enable Code Segment Register). You can also use it under program control to save data, using the `push` and `pop` instructions.

The user stack works exactly the same way, using the `pushu` and `popu` instructions but only under program control, which means that the user stack is not changed by the system. You may choose to use a separate space for your data, or to store them in the same stack as the return addresses.

Both stacks can independently be located either in RAM or in the register file. You select this using the SSP and USP bits in the MODER register (R235) for the system stack and the user stack, respectively. A low bit value selects a RAM stack, and a high bit value selects a Register File stack.

Since the stacks grow towards low addresses, the stack pointers must be initialized to the highest location plus one(3) of the space reserved to it. This location becomes the ”bottom” of the stack. When the stack is located in the register file, take care that it does not overwrite other data, in particular the registers located in groups 14 (0EH) and 15 (0FH). For this reason it is advisable to set the system stack pointer to the end of group 13 (0DH).

The last register of this group being R223, the instruction that sets the stack pointer will be:

```
   ld      sspr, #224 ; set stack pointer to one above end of group 13
```

Note 1: Using two separate stacks in the same kind of storage (memory or register) area is likely to consume more space than if a single stack is used. So most of the time, only one stack will hold both return addresses and arguments for functions. You can then use `pushu` and `popu` instructions manipulate data with the convenience of auto incrementing or decrementing of the pointer after each access.

As an example, the C language start-up files as described in Appendix B include the following statements:

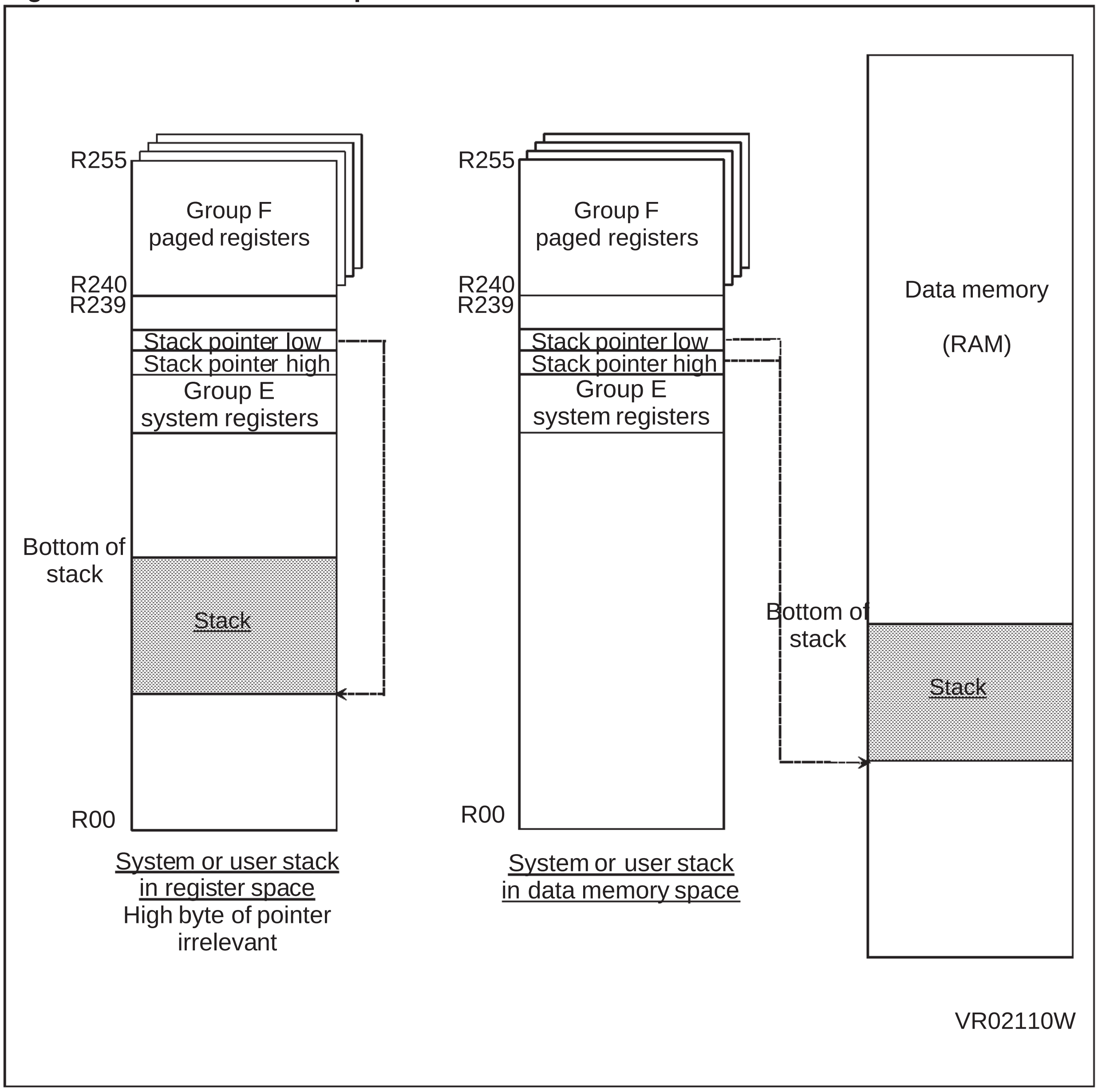
```
K_INITCLOCKMODE = 20h  ; MODER (R235) = both stack in memory + clock divided
   by 2
   ld      MODER,#K_INITCLOCKMODE ; init CLOCKMODE (both stack in memory)
   ldw     SSPR,#_stack_end      ; setup system stack
   ldw     USPR,#_user_stack_end ; setup user stack (not used actually)
```

The complete initialisation code is described in Section 5.9.3 Writing the Start-up File.

The following diagram illustrates the two options for locating the stack: in the register file or in memory.

(3) The push instruction decrements the stack pointer before writing the data, so setting it to the top location would never use this location.

Figure 11. Stack Location Options



3.3 INSTRUCTION SET

The ST9+ is said to be an 8/16-bit microcontroller. This means that although the size of the internal registers and the width of the data bus are 8 bits, the instruction set includes instructions that handle a pair of registers or a pair of bytes in memory at once. These instructions represent roughly one half of the total instructions, which means that the ST9+ can be programmed with the same ease as if it were advertised as a full 16-bit machine.

This is why it is well suited for C programming, as is illustrated in this book.

3.3.1 Overview

For a complete description of the instruction set, you should refer to the ST9+ Programming Manual. The aim here is to give you an introduction to the ST9+ instruction set and highlight some of its best features in terms of power and ease of programming.

Most instructions of the ST9+ exist in both byte and word forms. That is, they can operate on either 8 (byte) or 16-bits (word). The mnemonics of the word-instructions all end with a ”w”, as in the following examples:

Load	**Add**	**Subtract**	**Logical and**	**Logical or**	**Compare**	**Push**	**Pop**
ld	add	sub	and	or	cp	push	pop
ldw	addw	subw	andw	orw	cpw	pushw	popw

The new powerful instructions added to the ST9old are the CALLS, RETS, JPS instructions for far branching to change the program segment and the instructions used for C language applications, LINK, LINKU, UNLINK and UNLINKU. Moreover, all instructions have been optimized compared to ST9old.

3.3.1.1 Load instructions

Beside the classical load instructions found on most microprocessors, there are four special load instructions for moving data between two locations in memory.

One instruction to move data from data segment to data segment; `lddd`. This instruction allows you to post-increment the destination and the source index register at the same time. The ld instruction needs two instructions to do this. An example for moving a block of data would be:

```
    ld       rr0,#Source
    ld       rr2,#Destination      ; initialisation of the pointers
    ld       r5,#Num_loop          ; number of elements to move
loop:
    ld       r4,(rr0)+             ;  transfer of one byte
    ld       (rr2)+,r4
```

```
   djnz    r5,loop
```

The two `ld` instructions are coded using 6 bytes and execute in 24 cycles.

The same program with the `lddd` instruction:

```
   ld      rr0,#Source
   ld      rr2,#Destination     ; initialisation of the pointers
   ld      r5,#Num_loop         ; number of elements to move
loop:
   lddd    (rr2)+, (rr0)+       ; transfer of one byte
   djnz    r5,loop
```

The `lddd` instruction is coded using 2 bytes and executes in 14 cycles.

Here are the four possible data transfers:

Instruction	moves data from...	...to
lddd	data segment (uses the DPR0-DPR3 registers)	data segment
ldpp	program segment (uses the CSR register)	program segment
lddp	program segment	data segment
ldpd	data segment	program segment

These four instructions improve the performance of data block moves (frequently used in C programs).

As you can see in the table above, the data move can be between data and program segments. Here’s an example of a data move from a Data segment using the DPR register to a Program segment using CSR register:

```
   ld      rr0,#Source
   ld      rr2,#Destination     ; initialisation of the pointers
   ld      r5,#Num_loop         ; number of elements to move
loop:
   ldpd    (rr2)+, (rr0)+       ; transfer of one byte
   djnz    r5,loop
```

The data load with rr0 comes from the data segment selected by one of the four DPR register value depending on the rr0 value and then are stored in the program segment selected by the CSR register value.

(please refer to the MMU Section 3.1.7 for an explanation of data and program segments).

3.3.1.2 Test under Mask

These instructions, tm and tmw, perform a logical (bitwise) AND between the two operands, but do not store the result. They only set the Z and S bits of the flag register for later conditional jump on zero or sign. The mask is a value in which the bits that are set to 1 select the corresponding bits of the value to be tested for non-zero. As an example, in the following instruction:

```
    tm value, mask
```

If the mask is a byte whose binary value is 11000000, only the leftmost two bits of the unknown value will be tested, and a later branch if zero will be taken or not according only to these bits. As shown below, the same mask is used for two values that differ only by one bit:

0 0 0 1 1 0 1 0 1	Byte to be tested	1 0 1 1 0 1 0 1
1 1 0 0 0 0 0 0 0	Mask used for testing	1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0	Result of the logical AND operation	1 0 0 0 0 0 0 0
jump taken	Result of the ”jump if zero”	jump not taken

Two more instructions, tcm and tcmw, work essentially the same way, but they take the complement of the value to be tested before ANDing it with the mask, as follows:

```
    tcm value, mask
```

The same two cases will provide the following results:

0 0 0 1 1 0 1 0 1	Byte to be tested	1 0 1 1 0 1 0 1
1 1 1 0 0 1 0 1 0	Complement of the byte to be tested	0 1 0 0 1 0 1 0
1 1 0 0 0 0 0 0 0	Mask used for testing	1 1 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0	Result of the logical AND operation	0 1 0 0 0 0 0 0
jump not taken	Result of the ”jump if zero”	jump not taken

The jump would be taken if the byte to be tested had two 1’s in its most significant two bits, for example 11110101.

3.3.1.3 Push and Pop

Since there are two stacks, there are two kinds of push and pop instructions. The mnemonics push, pushw, pop and popw act on the system stack, which can be either in the register space or in the memory space. The mnemonics pushu, pushuw, popu and popuw act on the user stack, that can also be either in the register space or the memory space. The stack pointer used in each case is the SSPR or the USPR register pair respectively. The stack pointers are always decremented before writing on pushing, and they are incremented after reading on popping. Thus the stack pointer always points to the last byte written. This is worth knowing if you need to manipulate the stack contents.

The operands to be pushed can be a register, a pair of registers or an immediate value:

```
push r6
push (R120)
push #80
pushw RR100
pushw #1500
```

Pushing an immediate value is especially useful when you are programming in C.

A special push instruction is Push Effective Address. This instruction does not push the data itself, but the memory address of the data. For example:

```
pea 5(rr2)
```

This takes the contents of rr2, adds 5 and pushes the result onto the stack. This is widely used in C programming.

3.3.1.4 Multiply and Divide

The multiply instruction takes two byte operands and provides a word result. All numbers are treated as unsigned numbers (operands 0 to 255, result 0 to 65535). Though both operands are bytes, the first one must address a word to receive the result. The first operand should then reside in the low byte of the word, and the high byte, not used in the operation, will be overwritten. The flag register is affected but the state of the flags after the operation is meaningless.

To make a multiplication with signed number (operands -128 to 127) by unsigned number (operands 0 to 255) with a result in the range of -32768 to 32767 an example is given below:

```
   ld      r1,#signed_data
   ld      r4,#unsigned_data
   btjt    r1.7,neg
   mul     rr0,r4          ; rr0 = r1*r4, with r1 value a positive signed
                           ; number
   jp      end
neg:
   mul     rr0,r4          ; rr0 = r1*r4, with r1 value a negative signed
                           ; number
   sub     r0,r4           ; rr0=rr0-100h*r4
```

```
end
```

with the signed operand equal to 226=0E2h (means -30 for signed data) and the second unsigned operand equal to 0Fh (+15) the result will be -450 (0FE3Eh).

This eight bits signed by eight bits unsigned multiplication with sixteen bits signed result takes a maximum of 36 cycles.

There are two divide instructions.

The **div** instruction divides a word by a byte, and returns the quotient and the remainder as the low and the high bytes of the destination respectively. For example:

```
   ldw     rr0,#31184       ; rr0=#31184
   ld      r2,#201          ; r2=#201
   div     rr0,r2           ; rr0=1D9Bh, 1Dh=#29 and 9Bh=#155
```

This puts the value 155 in r1 (the quotient) and the value 29 (the remainder) in r0, and r2 still contains 201.

If the divider is greater than the dividend, nothing is done. If the divisor is zero, a trap is triggered that acts like an interrupt request, and uses the vector at locations 2 and 3 in program memory. It is up to you to write the appropriate code to handle this trap. Finally, the numbers to be divided should be such that the quotient be less than 256, that is, can be stored in a single byte. Otherwise, the results are undefined.

The usable result is only the data stored in r1 which is 155 (for the previous example), the remainder must be divided by the divisor (201) to give more precision (16-bits precision).

```
   ldw     rr0,#31184       ; rr0=#31184
   ld      r2,#201          ; r2=#201
   div     rr0,r2           ; rr0=1D9Bh, 1Dh=#29 the remainder and
                            ; 9Bh=#155 the quotient
   ld      r4,r1            ; r4=9Bh=#155
   clr     r1               ; rr0=1D00h
   div     rr0,r2           ; rr0=0BC24h, 0BCh=#188 the remainder and
                            ; 24h=#36 the quotient
   ld      r0,r4            ; rr0=09B24h, 09B24h means in fix-point
                            ; with the point in the 16-bits middle
                            ; 09B24h=155.140625 instead of
                            ; #31184/#201=155.1442786
```

In the best case the number of cycle to make a division of a word by a byte with 16-bits precision takes 80 cycles. To be used this program has to manage overflow and divide by zero.

The **divws** instruction performs one of the sixteen partial divides required to divide a double word by a word, so you need to write a subroutine to perform the division completely. An example subroutine is given in the ST9+ Programming Manual.

3.3.1.5 Bit Operations

Microcontrollers are often used for controlling inputs and outputs on a single bit basis, in order to read the state of a contact, switch a relay on or off, etc. Because of this and because the data is stored in bytes, instructions for bitwise manipulation of data are welcome.

The ST9+ provides instructions to load, and, or, exor, set, clear, complement and test single bits. These are bld, band, bor, bxor, bset, bres, bcpl, btset.

To designate a single bit in a byte, the notation .n is used. For example, r0.3 means bit 3 of r0. Here are examples of bit manipulation instructions:

```
bld     r0.2, r6.4      ; bit 4 of r6 copied to bit 2 of r0
bld     r0.3, !r6.0     ; complement of bit 0 of r6 copied to bit 3 of r0
band    r0.2, r0.3      ; r0.2 contains now (r6.4) and not (r6.0)
bor     r0.2, r2.7
bset    r0.0            ; bit 0 or r0 set to 1
bcpl r0.1; bit 1 of r0 is complemented
```

All the above instructions act on single working registers. If the source operand is preceded by '!', the complement of the source bit is used.

To test a bit, to condition a later jump, we have already described the tm and tcm instructions. There is another instruction, btset, that can act on either a single or a double working register, and that sets the Z bit of the FLAGR register if the designated bit is zero. After which, the bit is set to one. You can use this instruction in an interrupt service routine to test for a request and acknowledge it in a single instruction.

Warning. Don't use the bit manipulation instructions directly on bidirectional ports. To avoid unwanted modifications to the port output register contents use a copy of the port register, then transfer the result with a load instruction to the I/O port. (Refer to the ST90158-ST90135 datasheet Input/output bit configuration chapter for more explanation).

3.3.1.6 Test and Jump

The btjf and btjt instructions test if a bit is set or cleared respectively and branch to another program location if true. For example:

```
btjt    r1.5,Lampon
```

```
   bset    r1.5                  ; switch lamp on
Lampon:
   ...     ; continuation of the program
```

Two instructions are well-suited for implementing lookup tables. These are cpjfi and cpjti. They compare a byte in register with a byte pointed to by a register pair, and increment the pointer if the condition is not met. If the condition is met, the pointer is not incremented and the branch is taken. Example:

; Find the position of a letter in a text.

```
Message:.ascii ”This is a trial”

   ld      rr0,#Message          ; where to search
   ld      r2,#’t’               ; the character to search for

Search:
   cpjfi   r2,(rr0),Search       ; this is the search loop

   ...                ; here rr0 points to the 11th character of message
   ...                ; continuation
```

3.3.1.7 Far branch

As is explained in Section 3.1.7, the 4 Mbyte memory is a segmented memory. It is not possible to reach another segment with the common instructions CALL, RET and JP because they do not manage the CSR (Code Segment Register) register. The three new instructions CALLS, RETS and JPS manage it. Only 2 to 4 cycles are added to the common instructions.

3.3.1.8 Optimized C instructions

In C functions when a function is called, the compiler needs to push the variables in the user/ system stacks and to keep the return address location of the function inside the stack.

Therefore, a frame pointer is used, and 2 pieces of code named prologue and epilogue need to be added by the C Compiler at the beginning and at the end of the function respectively. The LINK and UNLINK instructions (LINKU, UNLINKU to use the user stack) are used to reduce the code overhead generated by the compiler inside the function. These instructions are automatically added by the C Compiler instead of prologue and epilogue (if option -mlink is specified).

The number of cycles gained by using these instructions is about 34 to 42 cycles and 8 bytes per called function.

3.3.2 Advantages when Using C

The ST9+ has been designed with high-level languages in mind. In particular, the instructions described above are of special interest to C programmers.

First, as a structured language, C typically uses the stack to pass arguments to functions, return values from functions, and store the local variables of the functions. An instruction such as:

```
    pushw #1500
```

pushes a constant integer value as the constant argument of a function. This is used in the following example:

```
/* define a function that has a single argument of int type */
void MyFunction ( int Param ) ;
    {
    /* body of the function */
    }

void main ( void ) ;
    {
    /* some code ... */
    MyFunction ( 1500 ) ;/* invoke this function with a constant argument */
    /* more code ... */
    }
```

Since C makes heavy use of pointers, the instruction:

```
    pea 4(rr2)
```

pushes the address of the 5th byte of a structure.

For example:

```
/* define a structured type */
struct sMyStruct {
    int N1 ;
    int N2 ;
    char C1 ;
    } ;

struct sMyStruct MyStruct ;/* create a variable of the above defined type
    */
```

```
/* define a function that has a single argument of type pointer to character
   */
void MyFunc ( char * Arg ) ;
   {
   /* body of the function */
   }

void main ( void ) ;
   {
   /* some code */
   /* invoke the function with the address of the character element of the
   structure */
   MyFunc ( &MyStruct.C1 ) ;/* &MyStruct.C1 = 4(rr2) if rr2 contains the
   address of MyStruct */
   /* more code */
   }
```

The lddd, ldpp, lddp, ldpd instructions are used for block copies such as assignments of structures, etc.

Powerful addressing modes such as indirect, indirect with increment or decrement, indexed shorten the code needed to access data even in structures or arrays. They also facilitate access to local variables created on the stack on entering functions. Working registers, that benefit from the most powerful instructions and addressing modes, are heavily used by the compiler. In fact, the GNU9 compiler does not always translate the source code as suggested above. There are optimization schemes that save execution time and/or memory by judiciously allocating the working registers, so that in many cases arguments are not pushed to the stack, but merely to an available working register.

3.4 INTERRUPTS

The interrupt system of the ST9+ is very powerful, and, in consequence, requires some thorough study to get the most out of it. However, it is worth learning it since it allows you to build very efficient programs with excellent interrupt response times.

The ST9+ interrupt system works the same as that of any microcontroller, except for two points that call for special attention: the vector mechanism and the priority mechanism[4].

[4] ST90158 - ST90135 Data Sheet; Interrupts § 4

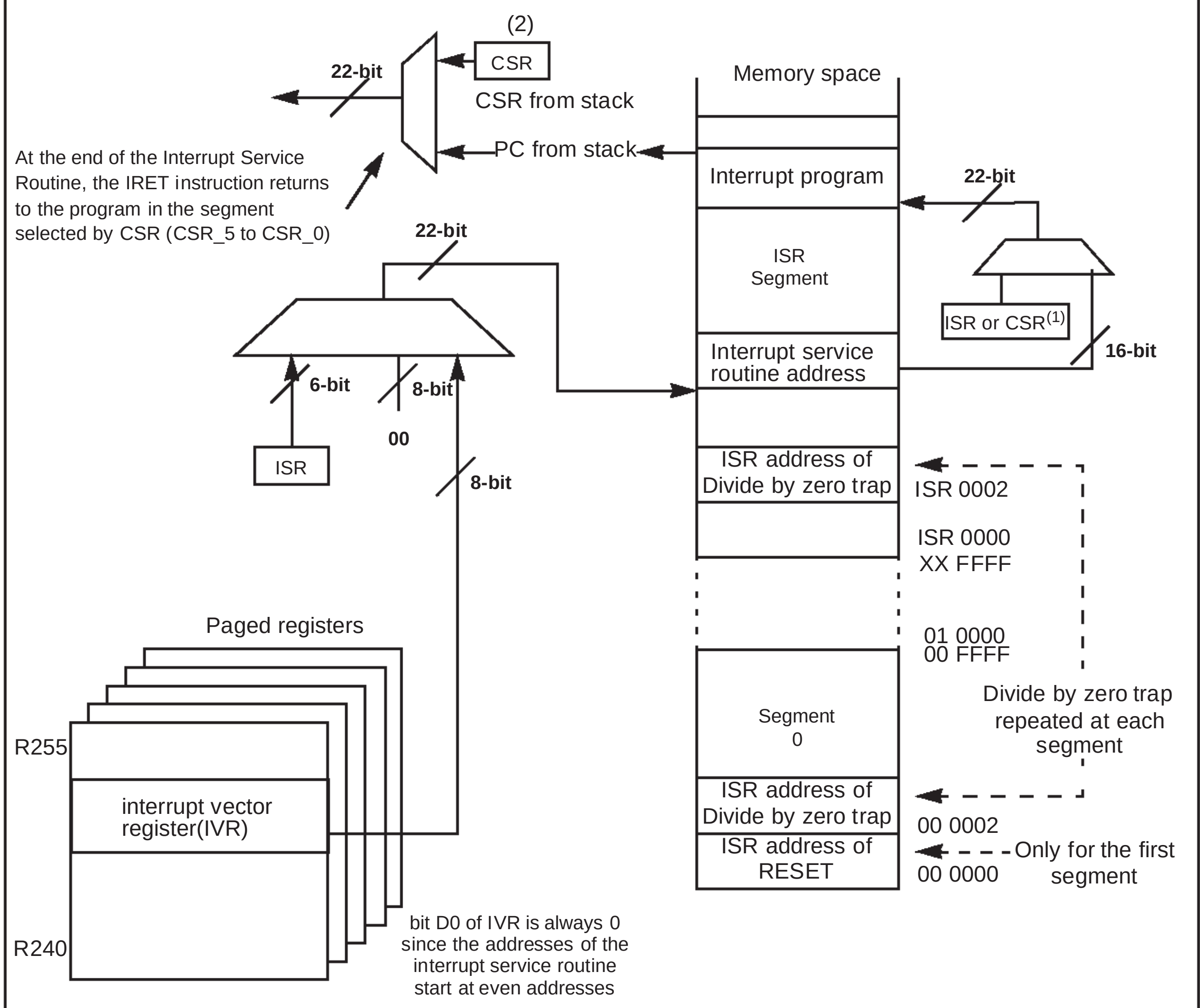
3.4.1 Interrupt Vectors

Unlike most microcontrollers, the ST9+ uses a two-level indirect interrupt vector system. This means that each peripheral able to generate interrupt requests has a vector register that points to a location in program memory (the vector array). This location contains the address of the start of the interrupt service routine. This allows each peripheral to generate several different interrupt requests: the peripheral vector register points to an array of pointers to routines, each routine responding to a different interrupt cause. All the pointers to interrupt processing routines except the Reset must be located in the first 256 bytes of the interrupt service routine (ISR) segment (one of the 64 code segments). The trap for divide by zero with the associated far call to the Interrupt service routine has to be repeated in all memory segments containing programs that perform division. Code may also reside in this 256-byte space, provided that it does not overlap with the interrupt vectors.

Figure 12 shows the complete mechanism of the 22-bit address construction from the interrupt which provides the Interrupt Vector Register value to the return from interrupt.

For each peripheral, the layout of the vector array is specified in the section related to its own Interrupt Vector Register.

Figure 12. Interrupt Vectors


(2)
CSR
22-bit
CSR from stack
PC from stack
Memory space
At the end of the Interrupt Service Routine, the IRET instruction returns to the program in the segment selected by CSR (CSR_5 to CSR_0)
Interrupt program
22-bit
ISR Segment
22-bit
ISR or CSR(1)
16-bit
Interrupt service routine address
6-bit
8-bit
00
ISR
8-bit
ISR address of Divide by zero trap
ISR 0002
ISR 0000
XX FFFF
01 0000
00 FFFF
Paged registers
Divide by zero trap repeated at each segment
Segment 0
R255
interrupt vector register(IVR)
ISR address of Divide by zero trap
00 0002
ISR address of RESET
00 0000
Only for the first segment
R240
bit D0 of IVR is always 0 since the addresses of the interrupt service routine start at even addresses

(1) If the ENCSR bit of EMR2 is set (Enable Code Segment Register), the CSR is saved in the stack and then loaded by the ISR. If it's reset, the CSR is not saved and only the ISR is used.

(2) If the ENCSR is set, the CSR is reloaded with the value saved in the stack when the interrupt occurs. If it's reset, CSR is used instead of ISR when the RETI instruction occurs (Return from interrupt).

As an example, let's take the ST90158 Multifunction Timer 0. The registers that define the MFT0 functions are all contained in pages 9 and 10 of register group 15 (0FH). Register 242, called the Interrupt Vector Register (IVR), holds the address of the beginning of the vector array in program memory. The IVR has the following bit layout:

T0_IVR (R242 page 9)

V4	V3	V2	V1	V0	W1	W0	'0'

Where V4-V0 (fixed by software) are the high bits of the low byte address memory where the vector for the first interrupt cause is located. Since there are three different interrupt causes, and the address of each interrupt service routine occupies two bytes, the IVR must be loaded with an address between 8 and 250 that is a multiple of 8 (i.e. the lower three bits are zero). The layout above shows two bits W1 and W0 (fixed by hardware), and a third bit that is permanently set to zero. The two bits W1-W0 code four different possible interrupt causes, as in the following table:

W1	**W0**	**Interrupt source**
0	0	Overflow/ Underflow event interrupts
0	1	Not available
1	0	Capture event interrupts
1	1	Compare event interrupts

When an interrupt occurs, the resulting value is an address that is the value written in IVR (here 40H), plus the value coded by the cause (if the cause is a capture event, W1-W0 are 10, thus the value 4).

This gives an even number, since the least significant bit is zero, as follows:

V4	V3	V2	V1	V0	W1	W0	0
0	1	0	0	0	1	0	0

So IVR points to the address XX 0044H (with XX the ISR segment number), at which must contain a word that is equal to the address in the ISR segment of the Interrupt Service Routine for that cause.

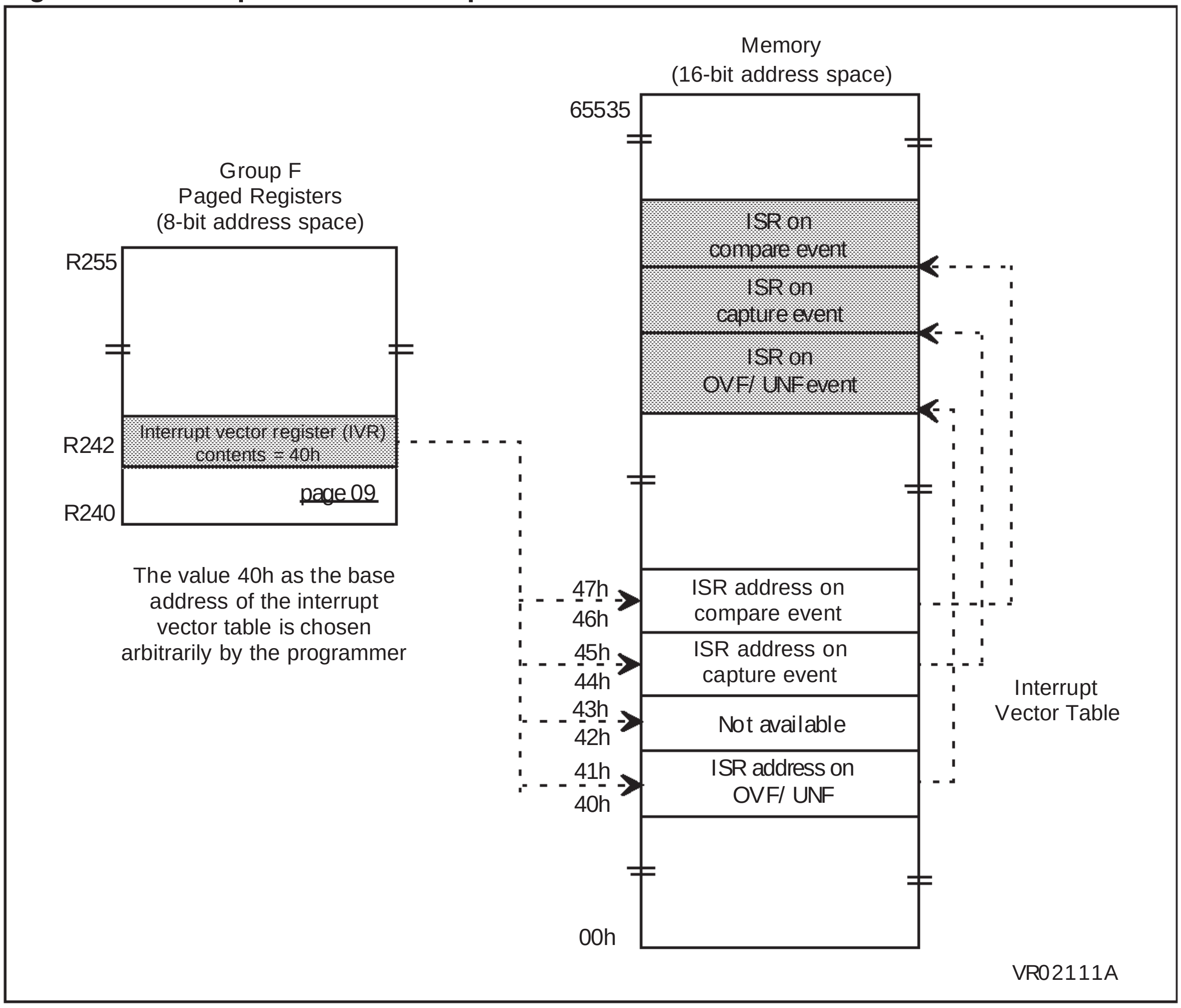
An example of the code for setting the IVR is:

```
void ConfigTimer0 ( void )
   {
   /* setting other registers... */

   SelectPage (T0C_PG) ;
   T0_IVR = (unsigned char)INTRELOADVECT ;/* Array of pointers to service
   routines in ROM */

   /* setting other registers, continued... */
   }
```

Figure 13. Interrupt Vectors: example with Multifunction Timer 0



The corresponding source code could be:

```
; This program uses the 3 interrupt possibilities of MFT0
.include « ST90158.inc »

; Constants
IT_MFT0_VECT := 40h

; Vector table
   .text
   .org    00h        ; (default address)
   .word   Reset      ; reset vector
   -       ; if needed divide by zero, NMI vectors
   -
   -
   .org     IT_MFT0_VECT
   .word    isr_ovf_unf ; ISR address on OVF /UNF
   .blkw    1           ; does not exist in MFT0 interrupt ; skip one vector
   .word    isr_capt    ; ISR address on capture event
   .word    isr_comp    ; ISR on compare event
   -
   -
   -
; somewhere in the initialisation code...

;MFT0 initialisation
mft0_init:
   spp      #T0C_PG     ; select MFT0 control register page
   -
   -        ; setting of some other registers...
   -
   ld       T0_IVR, #IT_MFT0_VECT ; pointer for the vector table
   -
   -
   -                    ; continuation of the program...
```

A similar scheme applies for all other peripherals, though the number of interrupt causes may vary, and thus the size of the pointer table.

Note: To summarise, the table of vectors to an interrupt service routine in ROM for a given peripheral is itself pointed to by the Interrupt Vector Register of this peripheral that must be set to the proper value. This gives you an unusual degree of flexibility: a peripheral may have different interrupt service routines at different times, without the need to add tests at the beginning of the interrupt service routine. Let's consider for example that we want to transfer a string of data from a peripheral. When the first byte of data comes in, we must initialise some variables to handle the string. Then, all subsequent transfers merely copy the data from the peripheral to memory and increment the pointer. Switching between these two modes is very easy. Initially, the IVR of the peripheral is set to the block of vectors that point to the interrupt service routine that serves the first time. This interrupt service routine changes the value of the IVR to another block of vectors and returns. The next interrupt will be automatically re-routed to the other, lighter, interrupt service routine. This reduces the execution time of this routine, since we do not need to test whether this is the first time that the interrupt service routine is called or not.

3.4.2 Interrupt Priorities

In any microprocessor-based system, there is a trade-off between the computational power of the main program and the interrupt latency time. Expressed simply, the less the main program is disturbed, the sooner it finishes its job. In the other hand, we often need to serve interrupt requests generated by the peripherals as quickly as possible. Since this is a trade-off, we need to find a compromise that gives both enough power to the main program while still keeping as responsive as possible to interrupts. It is likely that we will need to modify this compromise according to the current status of the program.

If we define that some interrupt requests are more urgent than others, we can define a priority or a hierarchy of interrupt requests. The main program itself, if given a priority level, should be considered as having the lowest priority (except sometimes when interrupts are undesirable).

In most cases, when the main program is running, it can be interrupted by any interrupt request. But once a request is being served, it most likely needs to continue undisturbed, unless a request from a higher priority level occurs. Then, the higher priority request is served until completion. The service routine then returns to the lower priority interrupt service routine, that terminates and eventually returns to the main program. In the ST9+, this behaviour is called Nested Mode.

Other types of behaviour may be required depending of the kind of processing. For this, the ST9+ interrupt system has two boolean parameters to select the way interrupts are handled, which allow four basic choices.

The priority mechanism is driven by the Current Priority Level parameter. At a given time, the part of the program being executed runs under a certain level. You can change this CPL by writing a different value in the core's Central Interrupt Control Register (CICR). You can assign Priority Level (PL) to each interrupt source. At initialisation time, this value is written in one of the control registers specific to the corresponding peripheral.

Notes: The PL is a three-bit word that ranges from 0 to 7. Note that 0 stands for a high priority and 7 for a low priority. These bits belong to one of the configuration registers of each peripheral.

When a peripheral requests an interrupt, the built-in interrupt controller compares the priority level of the interrupt request to the Current Priority Level. The interrupt is only acknowledged if its priority level value is strictly less than (higher priority) than the Current Priority Level value. This allows you to filter out interrupt requests according to their degree of importance or of urgency according to the current activity of the program. The Non-Maskable Interrupt input

(NMI) is hard wired with a higher priority than any level, and thus is acknowledged immediately in all circumstances.

The ST9+ offers two modes for managing interrupt priorities:

– Concurrent Mode

– Nested Mode

The difference between them is explained below.

3.4.2.1 Global Interrupt Enable Flag

In both modes, when an interrupt request is acknowledged, the Interrupt ENable (IEN) bit is cleared, preventing the interrupt service routine from being interrupted again until it is finished. If you need, you may prefer to keep it cleared for the duration of the routine or to set IEN at some place in the routine using the ei instruction. In the first case, if an interrupt request of a sufficient priority level is received, it will only be serviced as soon as the service routine currently running returns. In the second case, the same interrupt request is serviced as soon as both it occurs and the IEN bit is set. In short, the interrupt service routines may be re-interrupted or not, at will.

3.4.2.2 Concurrent Mode versus Nested Mode

Selecting either Concurrent Mode (that is automatically chosen on reset) or Nested Mode changes the way the Current Priority Level is managed.

In Nested Mode, the CPL is automatically set to the priority of the current interrupt service routine, and is reset to the previous value on return. This allows you handle the interrupt request according to priority at all times, since during the execution of the service routine for a given interrupt, only those interrupts whose priority is strictly higher than that of the one currently being served will be taken into consideration. Then if, as must normally be done, the IEN bit is set during the current service routine, it will be interrupted at once if an interrupt request of higher priority occurs. If the IEN remains cleared for the whole duration of the service routine, those interrupt requests will be served first after the current routine has returned.

Figure 14. Example with Nested Interrupts Enabled

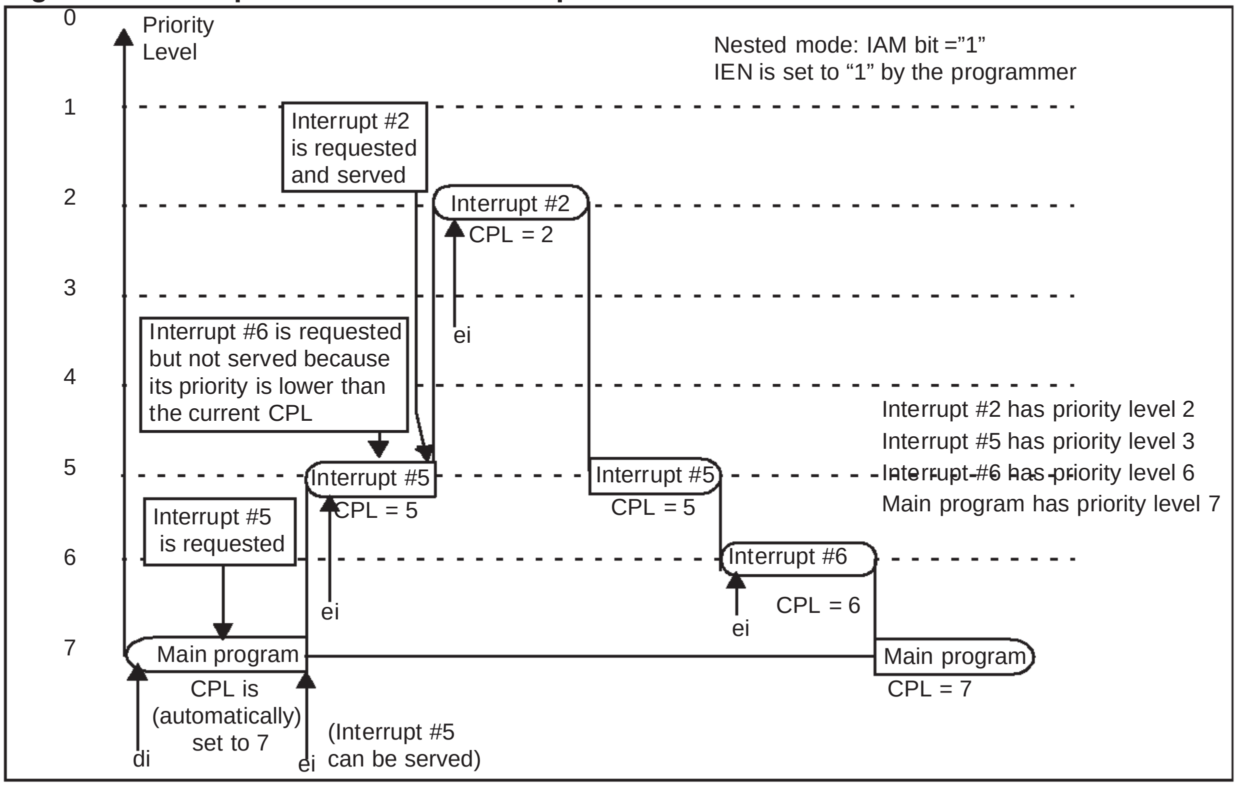
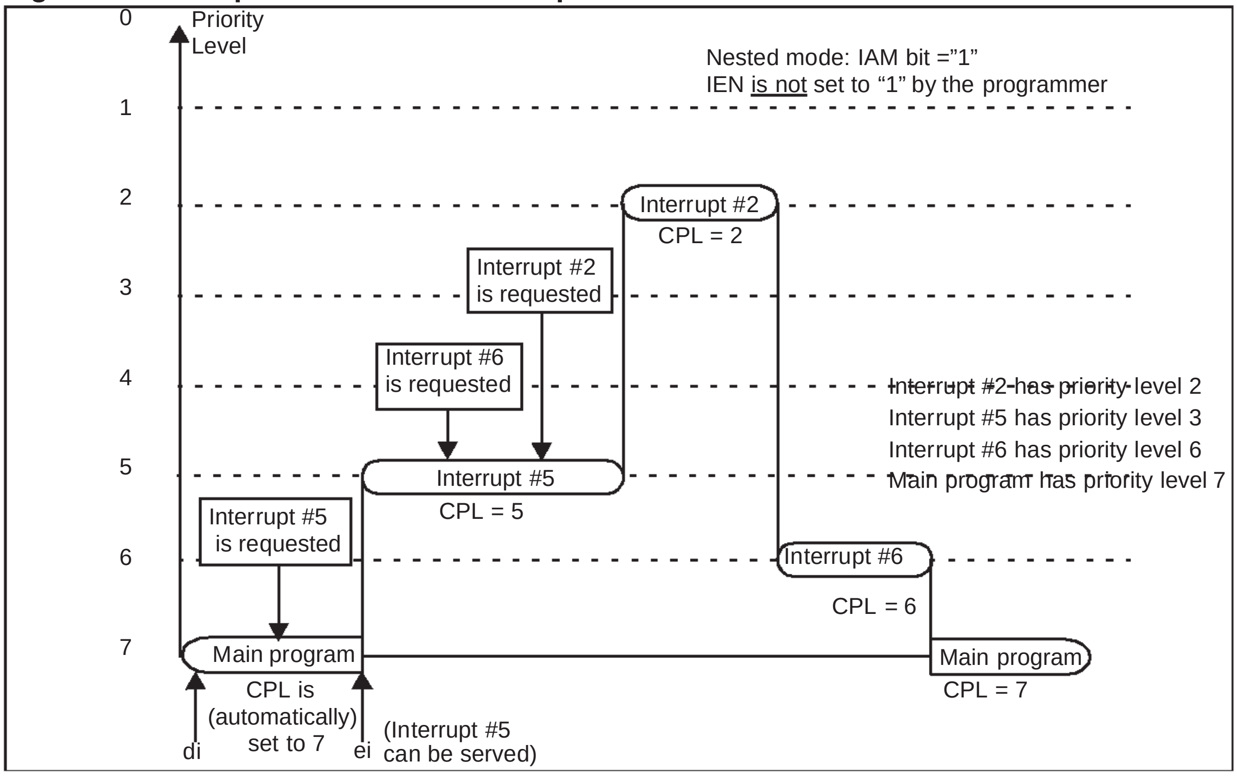
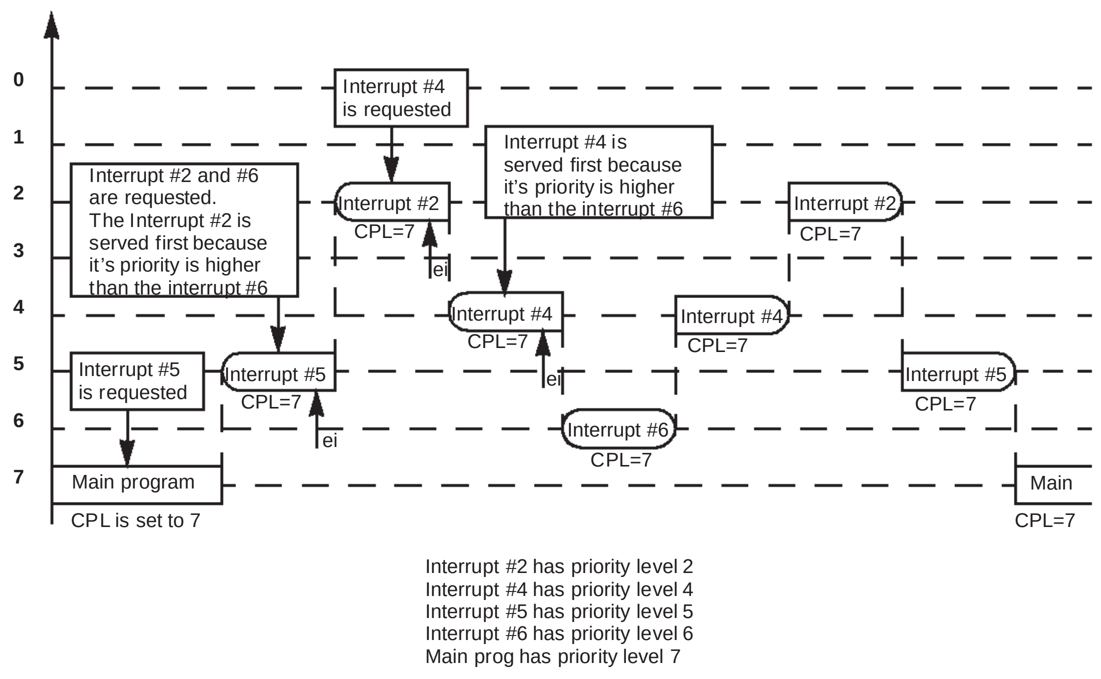

0
Priority Level
Nested mode: IAM bit ="1"
IEN is set to "1" by the programmer
1
Interrupt #2 is requested and served
2
Interrupt #2
CPL = 2
3
ei
Interrupt #6 is requested but not served because its priority is lower than the current CPL
4
Interrupt #2 has priority level 2
Interrupt #5 has priority level 3
Interrupt #6 has priority level 6
Main program has priority level 7
5
Interrupt #5
CPL = 5
Interrupt #5
CPL = 5
Interrupt #5 is requested
6
ei
Interrupt #6
CPL = 6
ei
7
Main program
Main program
CPL = 7
CPL is (automatically) set to 7
di
ei
(Interrupt #5 can be served)

Figure 15. Example with Nested Interrupts Disabled


0
Priority Level
Nested mode: IAM bit ="1"
IEN is not set to "1" by the programmer
1
2
Interrupt #2
CPL = 2
Interrupt #2 is requested
3
Interrupt #6 is requested
4
Interrupt #2 has priority level 2
Interrupt #5 has priority level 3
Interrupt #6 has priority level 6
Main program has priority level 7
5
Interrupt #5
CPL = 5
Interrupt #5 is requested
6
Interrupt #6
CPL = 6
7
Main program
Main program
CPL = 7
CPL is (automatically) set to 7
di
ei
(Interrupt #5 can be served)

In Concurrent Mode, the CPL is set exclusively by the programmer. You can change it if you need to shift the compromise mentioned above either towards main program efficiency, or towards short interrupt latency times. So you can have a high-priority service routine be re-interrupted by a low-priority interrupt request if the CPL has been set to a low priority and the IEN bit set. This mode gives you maximum flexibility, but it is the most difficult to use since you must keep track of every combination of interrupt requests to achieve the efficiency you expect from your program.

Figure 16. Example with Concurrent Interrupts Enabled


0
1
2
3
4
5
6
7
Interrupt #4 is requested
Interrupt #4 is served first because it's priority is higher than the interrupt #6
Interrupt #2 and #6 are requested. The Interrupt #2 is served first because it's priority is higher than the interrupt #6
Interrupt #2
CPL=7
ei
Interrupt #4
CPL=7
ei
Interrupt #5 is requested
Interrupt #5
CPL=7
ei
Interrupt #6
CPL=7
Interrupt #4
CPL=7
Interrupt #2
CPL=7
Interrupt #5
CPL=7
Main program
CPL is set to 7
Main
CPL=7
Interrupt #2 has priority level 2
Interrupt #4 has priority level 4
Interrupt #5 has priority level 5
Interrupt #6 has priority level 6
Main prog has priority level 7

Concurrent mode does not look like a reasonable way to handle interrupts if you enable interrupts in your interrupt service routine. It should be thought of as a way to fully control priorities through programming, if nesting priorities cannot meet your processing requirements. For example, let us consider the case when a timer produces a periodic interrupt that outputs some data on an external digital to analog converter. The requirement is that the new data be output at the very time of the interrupt, so as to reduce the jitter (or parasitic frequency modulation) that would compromise the spectral purity. Then, once the data is output, the interrupt service routine does some processing to make or get the data for the next interrupt. The latter part of the processing is much less critical in terms of execution time, provided it is finished before the next timer interrupt. Using Concurrent mode, you can assign that interrupt the highest priority so that it will be served immediately, then re-enable the interrupts and, if needed, give it a priority level as appropriate. The service routine will then allow other interrupts to gain control, at the expense of delaying its own completion.

Note 1: In practice, concurrent mode does not differ much from nested mode if the interrupts are not re-enabled during an interrupt service routine. Concurrent Interrupts Disabled looks like Nested Interrupt Disabled with the difference that the CPL is changed only by software if it’s necessary. The interrupt requests pending while the interrupt service routine is executing, will only be serviced after the current service routine returns.

Note 2: An interrupt with priority level 7 will never be served.

3.4.3 External Interrupt Unit

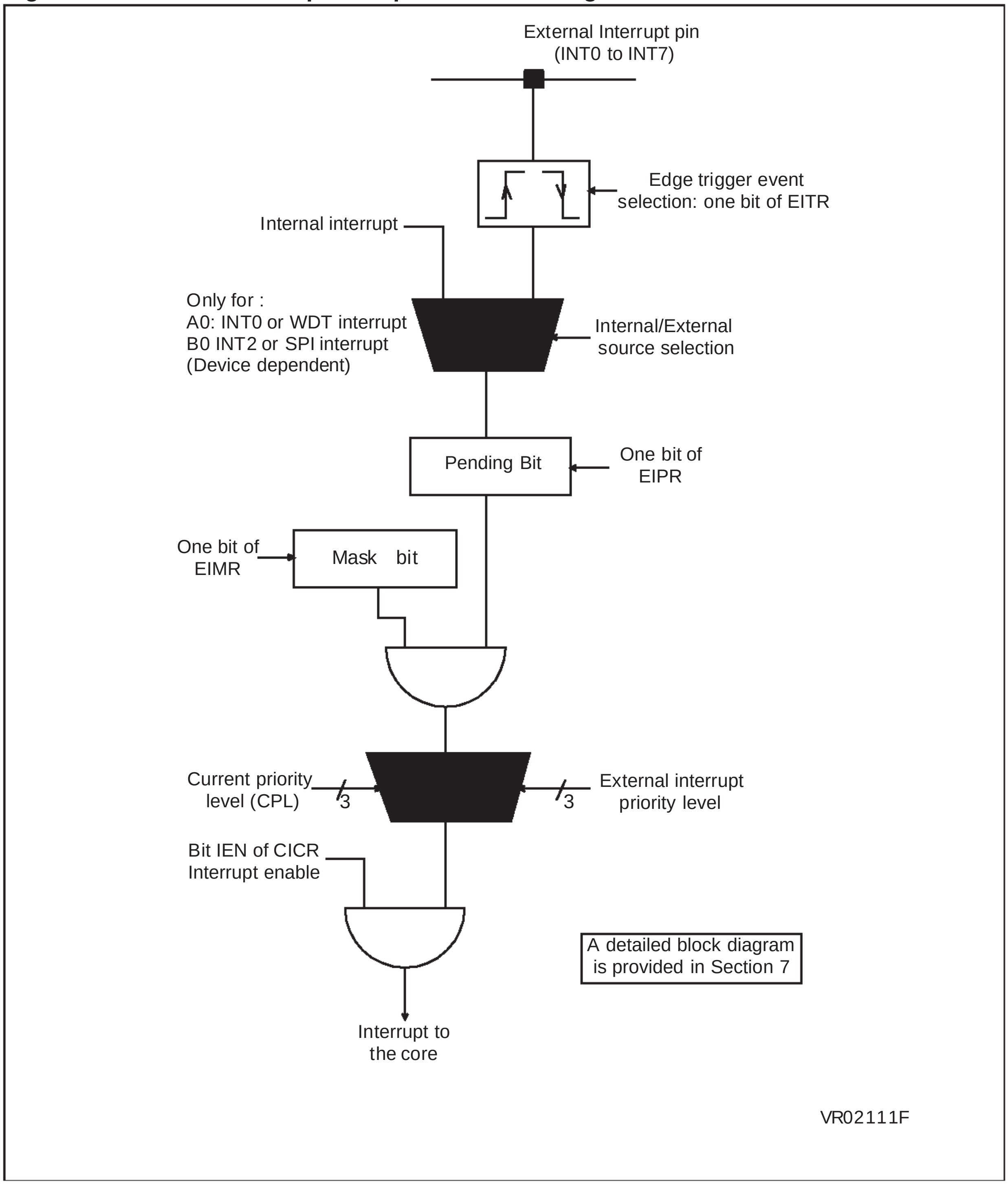
This is a functional block that can receive interrupt requests from up to eight external pins, and also from some internal devices such as the Watchdog Timer, the Serial Peripheral Interface, etc. It allows selecting the active edge for each, and the priority for each of the four pairs. In addition, an NMI pin can be programmed as maskable or non-maskable. It is maskable on reset, and once set to non-maskable, it cannot be set maskable any more until the next reset. This works as follows:

3.4.3.1 Maskable External Interrupt Pins

These are eight external inputs you can set individually to rising-edge or falling-edge sensitive, and masked. You assign priorities by pairs, making four groups with different priorities. Within each group, the two interrupt requests have two successive priorities. For example, if you set group C to priority 2, the INT4 input will have priority 2 and INT5 will have priority 3 (which is lower).

The simplified block diagram is the following:

Figure 17. External Interrupts Simplified Block Diagram



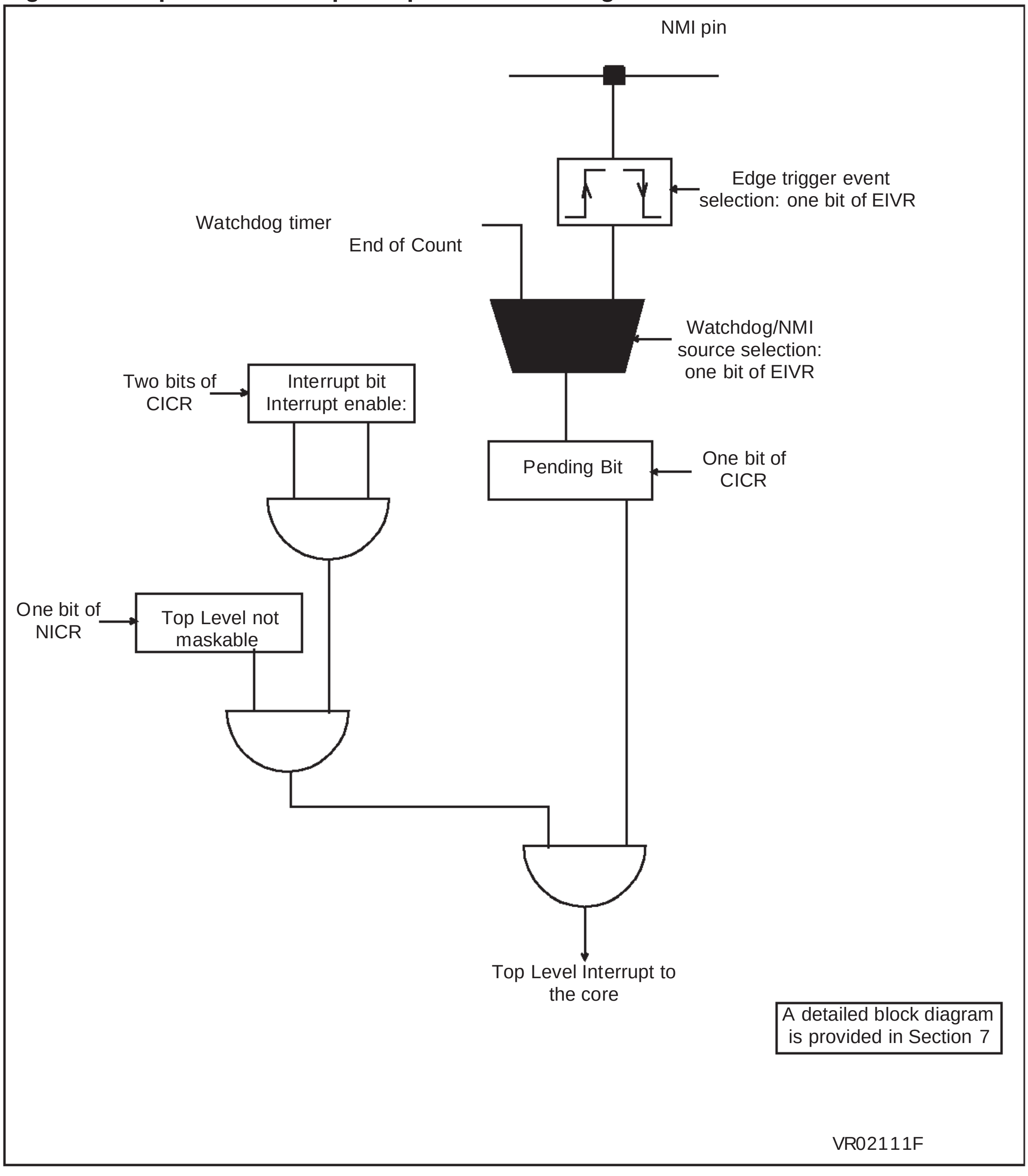
3.4.3.2 Top-Level Interrupt

The Top Level Interrupt can have two sources: either the NMI pin, or the watchdog end-of-count(5). This interrupt level has a special feature that allows you to mask it like any other interrupt, or to make it a real non-maskable interrupt. For this, the TLNM bit (see Figure 18) can remove the effect of the mask. Once set, it cannot be reset, thus preventing this interrupt from being accidentally masked even in the case of a program failure. In any case, the Top Level Interrupt uses the third fixed vector at addresses 4 and 5 in program memory. As the name implies, it has a fixed priority that is higher than any other interrupt request. Though it does not clear the IEN bit when acknowledged, unlike all other interrupt requests, its service routine cannot be interrupted by any cause, including the Top Level Interrupt itself.

The simplified block diagram is the following:

(5) See Section 4.3 on the Watchdog Timer.

Figure 18. Top-Level Interrupt Simplified Block Diagram


NMI pin
Edge trigger event selection: one bit of EIVR
Watchdog timer
End of Count
Watchdog/NMI source selection: one bit of EIVR
Two bits of CICR
Interrupt bit
Interrupt enable:
Pending Bit
One bit of CICR
One bit of NICR
Top Level not maskable
Top Level Interrupt to the core
A detailed block diagram is provided in Section 7
VR02111F

3.4.3.3 External Interrupt Vectors

There are eight external interrupt causes. They are each connected to an external input that is the alternate function of an I/O port. Some of these are shared with other causes in an exclusive manner, i.e., the INT0 pin is multiplexed with the Watchdog/Timer interrupt request, and the INT2 pin is multiplexed with the Serial Peripheral Interface interrupt request. Each cause is associated with a separate vector in Program Memory. All vectors are contiguous, generating an array of 8 vectors starting with the INT0 vector, and ending with the INT7 vector. This array may be freely located in program memory between addresses 8 to 240 (0F0h) in program memory.

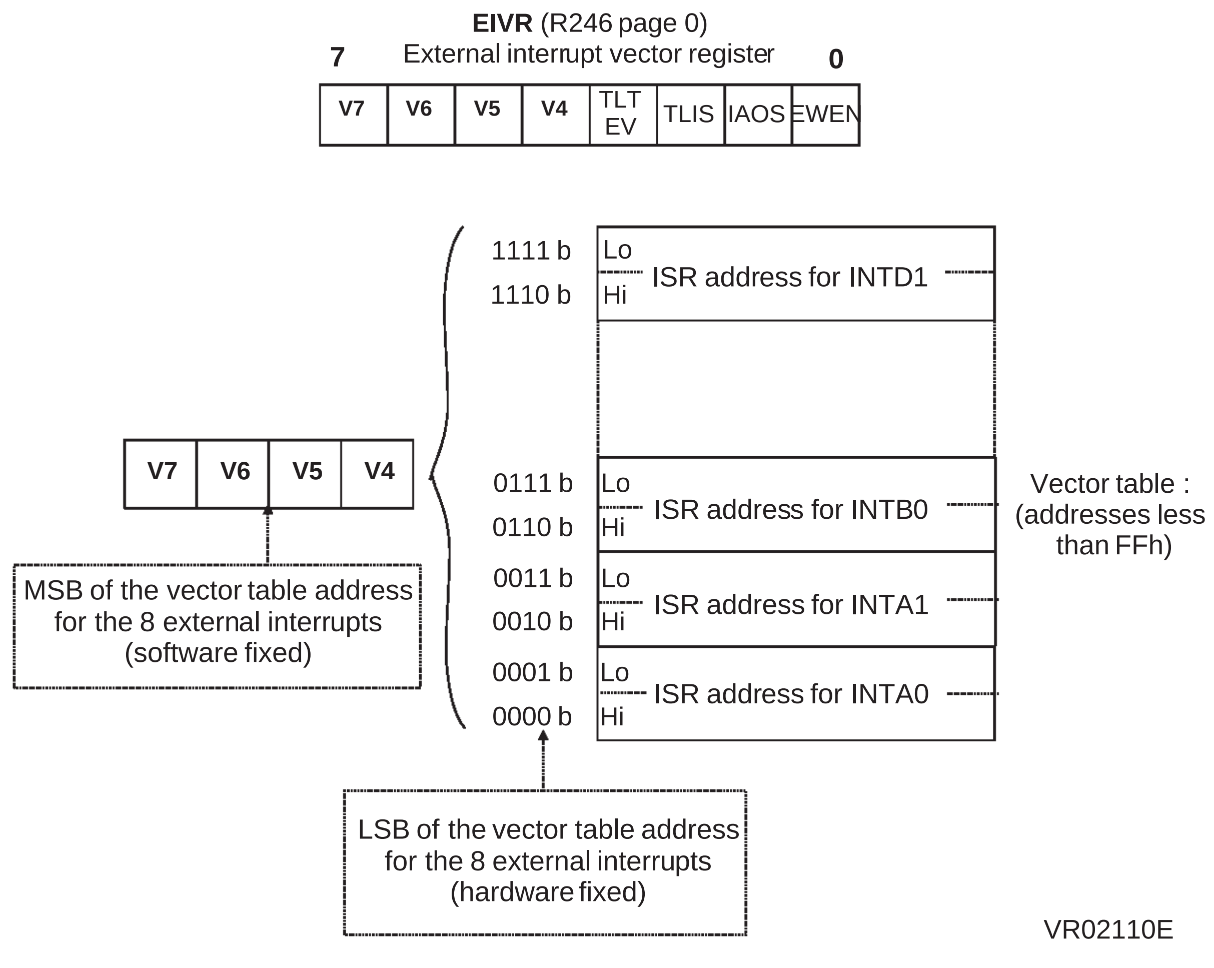
These interrupt causes are grouped by pairs, and are given new names inside the interrupt controller, as shown in the following table:

Interrupt source	Interrupt cause
INT 0	INT A0
INT 1	INT A1
INT 2	INT B0
INT 3	INT B1
INT 4	INT C0
INT 5	INT C1
INT 6	INT D0
INT 7	INT D1

You can independently assign a priority to each pair (A, B, C, D), with levels that are multiple of two, i.e. you can set them to priority levels 0, 2, 4, or 6. In each pair, the cause bearing the figure zero assumes this priority, and the cause bearing the figure 1 assumes the next level above. For example, if you assign level 4 to pair C, this means that INTC0 will have level 4 and INTC1 level 5. You set this in register EIPLR (R245 page 0), where each group of two bits gives the level of the corresponding interrupt cause.

The interrupt vectoring is summarised in the table below:

Figure 19. External Interrupt Vectors


EIVR (R246 page 0)
External interrupt vector register
7
0
V7
V6
V5
V4
TLT EV
TLIS
IAOS
EWEN
1111 b
1110 b
Lo
Hi
ISR address for INTD1
0111 b
0110 b
Lo
Hi
ISR address for INTB0
0011 b
0010 b
Lo
Hi
ISR address for INTA1
0001 b
0000 b
Lo
Hi
ISR address for INTA0
Vector table : (addresses less than FFh)
V7
V6
V5
V4
MSB of the vector table address for the 8 external interrupts (software fixed)
LSB of the vector table address for the 8 external interrupts (hardware fixed)
VR02110E

3.5 DMA CONTROLLER

3.5.1 Overview

One of the most important advantages of the ST9+ is its ability to handle input/output data flow without using core instruction cycles. This is made possible by the built-in DMA controller. Once properly initialised, it allows peripherals to exchange data either with memory or the register file, with no more use of the core resources than the stolen memory cycles strictly needed to transfer the data.

To see how much faster DMA is than the simplest interrupt service routine, let us compare the execution times.

Let us assume that the data comes from the serial port. The simplest interrupt routine is the following:

```
GetOneByte: ; interrupt latency:                      22 cycles
   push    PPR              ; save current page       8
   pushw   rr0              ; save rr0                10
   spp     #SCI_PG          ; change register page    4
   ldw     rr0, POINTER     ; get pointer             12
   ld      (rr0)+, S_RXBR   ; move data               12
   ldw     POINTER, rr0     ; store pointer           14
   popw    rr0              ; restore rr0             10
   pop     PPR              ; restore current page    8
   iret                     ; return                  16
           ; -------------------------------------------
           ; Total:                                   116
```

With an internal clock of 25 MHz, this corresponds to an execution time of 4.64 µs. In contrast, the DMA cycle time for the transfer of one byte from a register to a register file takes only 8 cycles (16 cycles to the memory), that is 0.32 µs. The DMA feature saves you using valuable core processing power for simple tasks like storing an input byte to memory. For example, if a continuous flow of data is input at 19200 bits per second, the interrupt service routine would consume 1.11% of the total cycles of the core, as compared to the 0.0767% with the DMA solution. Since the DMA is built-in and works with most of the peripherals, it is a good idea to use it even for slow transfers.

The DMA uses the Segment mechanism to address 64 Kbytes along the linear 4 MBytes. See Section 3.1.7 for more explanation.

3.5.2 How the DMA works

The DMA consists of a transfer between memory or register file and a peripheral, in either direction. Assuming the peripheral is configured to handle externally supplied data or to provide data to external circuits, two steps are needed for a transfer to occur.

The transfer must be requested by some event or condition.

A mechanism must handle reading the data from one part and writing to the other part.

The term DMA transfer represents the transfer of a single byte of data. Usually, more than one byte is transferred and the transfer occurs in bursts. Thus, a third step is involved:

A mechanism that counts the transfers and stops them when the count is finished.

To the programmer, these mechanisms appear as registers to be properly initialised. The three steps are handled as follows (see Figure 20):

3.5.2.1 Transfer Requests

The DMA transfer is requested in exactly the same way as an interrupt is requested. According to its type, the peripheral concerned sends a request based on an external event such as a character received or transmitted by the Serial Channel Interface. You configure this in the registers belonging to the peripheral involved. Special bits in the registers indicate that the peripheral ready status, instead of requesting an interrupt, requests a DMA transfer.

3.5.2.2 Transfer Execution

A DMA transfer can involve a memory location or a register in the register file. In any case, the transfer requires an address and a counter. This address and counter are stored in registers which you can define anywhere in the register file. The address register value is automatically incremented after each transfer and the counter register value is decremented so that the next transfer will involve the next address and this mechanism will continue until the counter is equal to 0. Depending on whether the transfer addresses memory or the register file, there are two cases.

If the transfer addresses the register file, the address register and the counter register are single registers (a byte is enough to address 256 registers). The address of this address register is stored in one of the peripheral's registers, named DCPR. The low bit (RM) of this register is set to zero indicating that the register file is involved in the transfer. The address register must have an even address to address the DMA address, the next register storing the DMA transaction counter.

If the transfer addresses the memory, the registers that hold the address and the counter must be two double registers. In this case, the DMA address is pointed by the DAPR pointer register and the DMA transaction counter is pointed by the DCPR pointer register. The low bit of the

DAPR indicates whether the memory segment is pointed by DMASR or ISR (see Section 3.1.7). The DMA address and the DMA transaction counter are not necessarily consecutive.

3.5.2.3 Transfer Termination

A burst terminates when the count of transfers reaches a predefined value. To set this up, you set a counter register in the register file that holds the count of transfers to execute at the start of a burst. Each transfer decrements it, and the DMA mechanism is stopped when the counter reaches zero.

The way DMA controller terminates the transfer differs from one peripheral to another, but typically it consists of resetting the bit in the configuration register that tells the peripheral to issue DMA requests instead of interrupt requests. Then, on the last transfer, when the transfer counter reaches zero, it toggles the DMA/Interrupt request bit so that the peripheral issues an interrupt request again. If this request is unmasked, you should vector it to an interrupt service routine that handles the DMA termination.

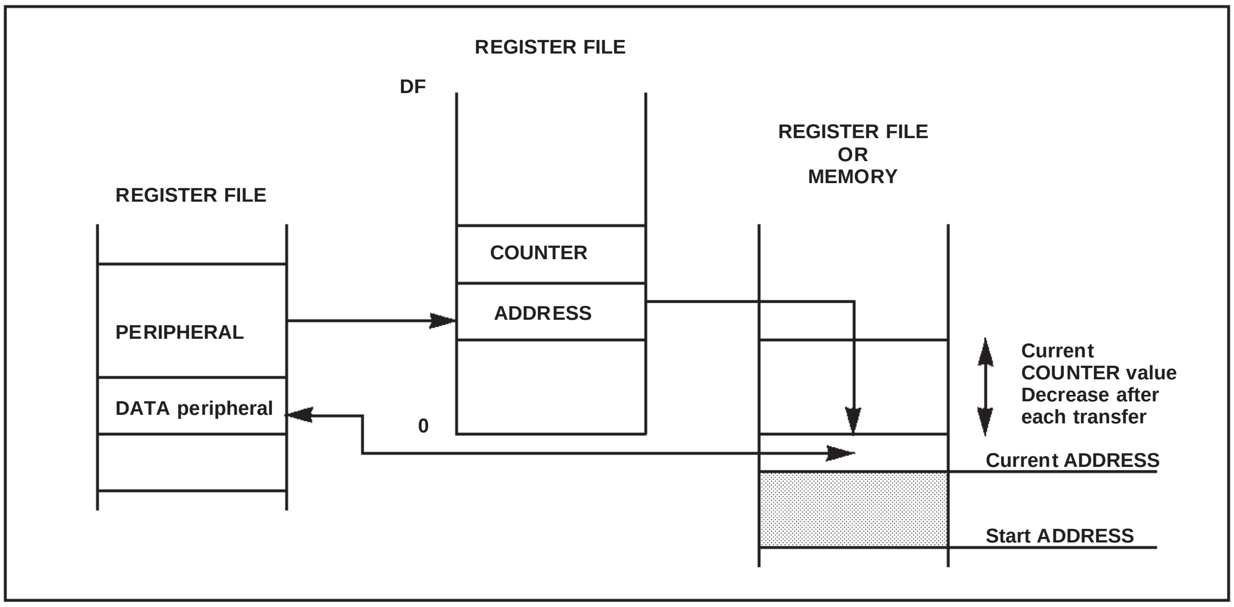
To summarise:

For register transfers, the DCPR of the peripheral points to a user-defined pair of registers that holds the address and the count. Register DAPR is unused. For memory transfers, the DAPR points to the user-defined register pair that holds the address. The DCPR register points to the user-defined register pair that holds the count.

Some peripherals, such as the Multifunction Timer, have even a double pair of DAPR/DCPR registers. Only one pair is used at a time. In so-called Swap Mode, this allows the timer to use one data buffer for output and another buffer for input. The pair of registers used is automatically changed when one transfer burst terminates, allowing a continuous data flow between the program and the peripheral, with minimum data handling overhead.

In addition, the MFT has 16-bit registers. Transfers imply two byte transfers for each register. This is ensured by a mechanism that gives the DMA the highest priority as soon as the first byte is transferring. This guarantees that the second byte will also be transferred in the shortest delay, even if other DMA requests become pending while the transfer is in progress.

Figure 20. DMA Data Transfer



3.6 RESET AND CLOCK CONTROL UNIT (RCCU)

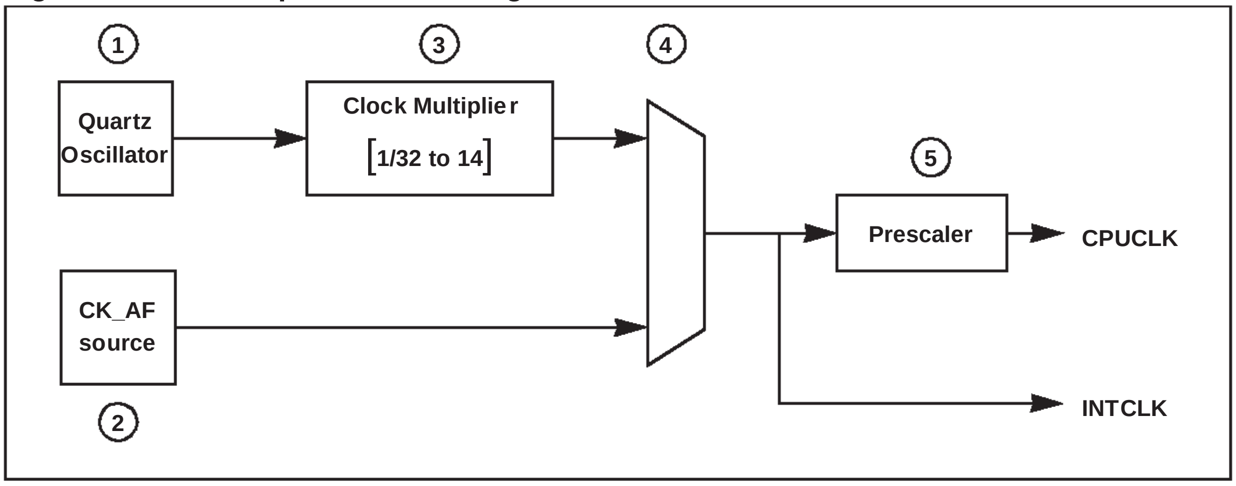
The **RCCU** is composed of the Clock Control Unit (**CCU**) and the Reset and Stop Manager.

3.6.1 CLOCK CONTROL UNIT

The **CCU** generates the peripheral clock **INTCLK** and the CPU clock **CPUCLK**. It is a useful clock generator with low power function and low external frequency oscillator to reduce electromagnetic emission.

The schematic can be reduced as in Figure 21

Figure 21. CCU simplified block diagram:



The CCU is composed of 5 main blocks:

BLOCK	COMMENTS
1	Quartz oscillator gives the main frequency generator. The frequency is in the range of 3 to 5 MHz.
2	External clock for very low power consumption with a very low frequency.
3	The Clock Multiplier can reduce or increase the input frequency by using prescaler and PLL.The increase is for the normal use and the decrease for low power consumption.
4	The selector aims the clock coming from the Clock Multiplier for normal or low power use or aims the clock CK_AF for a very low power consumption.
5	The prescaler is used also for low power consumption. The INTCLK is not change giving a high frequency to the internal peripheral. This allows the user to slow down program execution during non processor intensive routines.

3.6.1.1 The Clock Multiplier

The Clock Multiplier is a part of the Figure 23

The advantage of the Clock Multiplier is that it can produce a variable clock frequency depending on the CPU needs.

Since the input clock to the PLL circuit requires a 50% duty cycle for correct operation, the divide by two should be enabled (DIV2 bit of MODER register set) if the PLL is enabled. It is necessary when a crystal oscillator is used, or when the external clock generator does not provide a 50% duty cycle. In practice, the divide-by-two is virtually always used in order to ensure a 50% duty cycle signal to the PLL multiplier circuit.

The Clock Multiplier output is one of the CLOCK2, CLOCK2/16 or CLOCK2*PLLMUL/(DX+1) (PLLMUL is the PLL multiplier coefficient) frequencies.

The two frequencies CLOCK2 and CLOCK2/16 are for low power consumption or reduce power consumption, depending on Wait For Interrupt instruction or not (refer to the flow chart Figure 22).

The PLL has four clock multiplier factors (6,8,10 and 14) controlled by the two bits MX0 and MX1 in the PLLCONF register. The clock divider is controlled by three bits DX0:2 in the PLLCONF register for seven rates which are 1/(DX+1). Setting DX2:0=7 turns OFF the PLL to reduce consumption.

When you switch on the PLL you have to allow a delay for the PLL to lock.

Figure 22. INTCLK and CPUCLK Flow Chart.

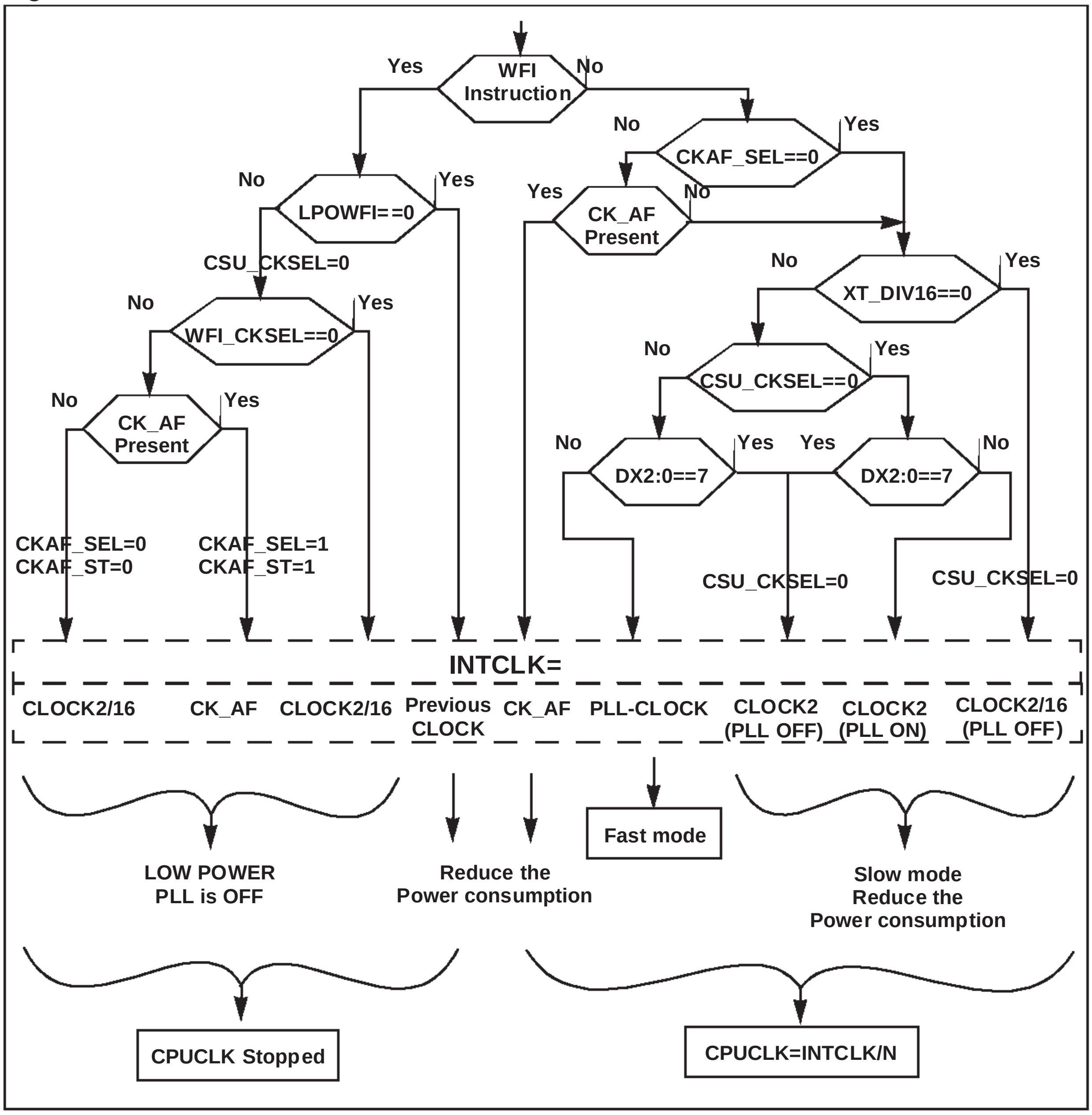
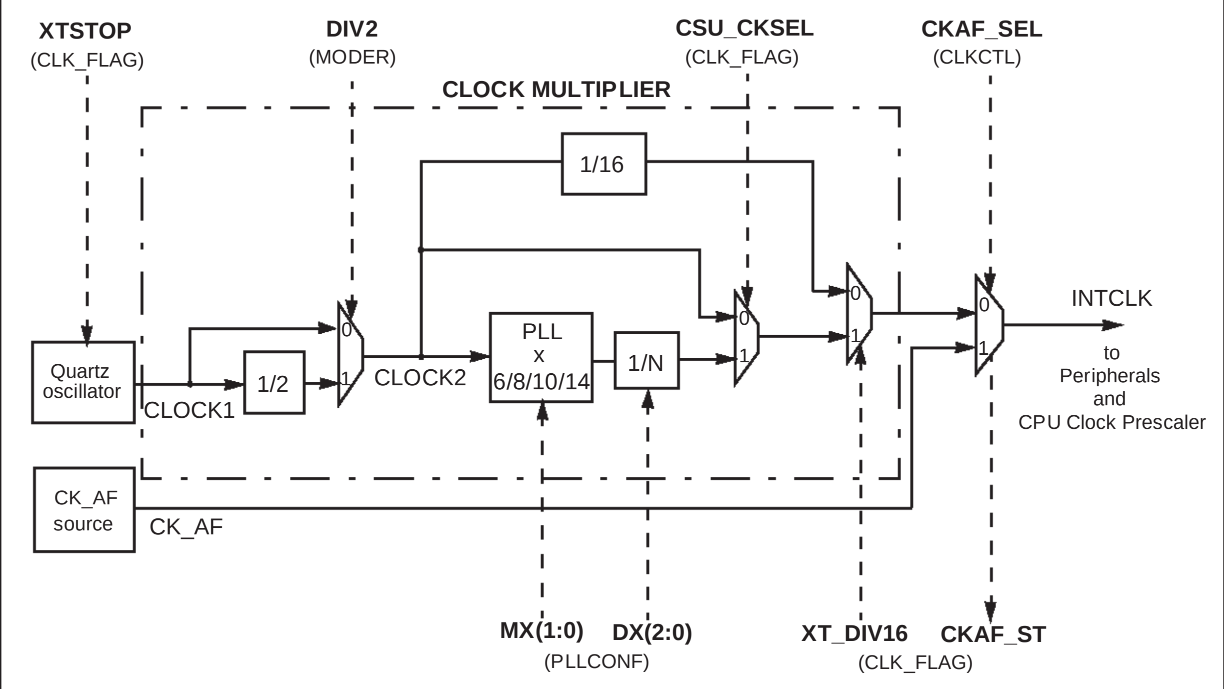


Figure 23. Clock Control Unit Programming



Wait for Interrupt and Low Power Modes:

LPOWFI (CLKCTL) selects Low Power operation automatically on entering WFI mode.
WFI_CKSEL (CLKCTL) selects the CK_AF clock automatically, if present, on entering WFI mode.
XTSTOP (CLK_FLAG) automatically stops the Xtal oscillator when the CK_AF clock is present and selected.

3.6.1.2 The CK_AF source.

When you execute a Wait For Interrupt (WFI) instruction using the Clock Multiplier with the output clock CLOCK2/16, the power is put in Low Power mode. To reduce this power further you have the possibility to slow down the INTCLK frequency by using an external clock source CK_AF. The CK_AF clock will be selected if the WFI_CKSEL bit in the CLKCTL register is set and if CK_AF is present.

The CK_AF source can also be used in run mode (no WFI) to reduce power consumption if CKAF_SEL is set and CK_AF is present.

3.6.1.3 Low Power with frequency slow down

When the PLL has been frozen by a WFI instruction you need a delay after the interrupt to wait for the PLL to lock. To avoid this delay but not to lose the Low Power consumption, you have two choices which are:

– To use a WFI instruction with LPOWFI reset and initialize INTCLK to CLOCK2 with XT_DIV16 bit set (CSU_CKSEL register). When the WFI occurs, CPUCLK is stopped and INTCLK doesn’t change.
– To not use the WFI instruction. Set XT_DIV16 bit of CSU_CKSEL register. Initialize DX2:0 of PLLCONF to 6 to divide the PLL output clock by 7 which is the maximum. Initialize the CPUCLK clock prescaler to 7 to divide INTCLK by 8.

3.6.1.4 Example using the Watchdog Interrupt

To show the different steps from Low Power mode to normal speed, here’s an example using the Watchdog Interrupt to wake up the ST9+.

The Low Power clock is CLOCK2/16 which has the PLL switched off.

The RCCU initialization is done by the INIT_PLL routine from the RCCU.C file.

```
void INIT_PLL(void)
{
   unsigned int loop;
   spp(RCCU_PG);
   PLLCONF = Cm_mul6 + Cm_div1;/* initialize the PLL, Mul. by 6, div. by 1 */
   CLKCTL &= ~Cm_ckafsel;
   for (loop = 0; loop< 50 ; loop ++); /* WAIT 500µs to stabilize the PLL*/
   CLK_FLAG |= (Cm_csucksel + Cm_xtdiv16);
}
```

The main program is FILE0.C and contains an infinite loop initializing the Low Power mode with the WFI instruction. The counter variable counts the number of interrupts which is 1 per second.

```
#include ”rccu.h”
/* declare prototypes of external functions used in main  */
extern void InitWDT_Rect();
extern void InitIntr();
extern void INIT_PLL();
#define WFI() asm ( ”wfi” ) ; /* Wait For Interrupt */
/* Variable */
int counter;
/* main function, launched by the startup program CRT9F.ASM */
void main (void)
{
 counter = 0;
```

```
   InitWDT_Rect();
   InitIntr();
   INIT_PLL();
   CLKCTL|=Cm_lpowfi;
   while (1){
     WFI();
     counter++;
   }
}
```

The two functions `InitWDT_Rect`() and `InitIntr`() are Assembler functions and they are called with a local CALL instruction because all the routines are placed in the first segment.

`InitWDT_Rect`() initializes the Watchdog timer for 1 interrupt per second. `InitIntr` () enables the watchdog interrupt. These two routines are in the file wdtrect.asm:

```
;============================================================
;=  Programming the WDT as periodic interrupt of 1 second using  =
;=  the top level interrupt to reload the WDTR                   =
;============================================================
   .include  ”system.inc”  ; System register
   .include  ”page_0.inc”  ; Page 0 registers
   .include  ”rccu.inc”  ; RCCU register
;*** For a WFI instruction with XTAL=4Mhz, DIV2 of MODER is set,
;*** XT_DIV16 bit of CLK_FLAG is reset,
;*** the Watchdog clock = INTCLK/4 = 4Mhz/32/4 = 31250 Hz.
;*** For a WATCHDOG interrupt every 1 second and a
;*** WATCHDOG prescaler (WDTPR ) of 128 the WDTR counter = 31250/128 = 244
LONG = 244  ; off time = 1 second  (high level on WDTOUT pin)
; Reset and interrupt vectors.
   .text
; InitWDT_Rect and InitIntr are called from the main routine.
; ReloadWDT is stored in CRT9S.ASM at the Watchdog Interrupt Vector
; Register.
   .global InitWDT_Rect,InitIntr,ReloadWDT
   .extern IntVect
; Initialization of the WDT as a rectangular signal output
InitWDT_Rect:
   spp     #WDT_PG               ; select watchdog page
   ld      WDTPR,#127       ; prescaler 1/128
   ldw     WDTR,#LONG       ; For 1 second in low power mode
; the Watchdog timer is in continuous mode.
   ld      WDTCR,#WDm_stsp+WDm_outmd+WDm_wrout+WDm_outen
   ret
; Initialization of interrupt system
InitIntr:
   spp     #EXINT_PG             ; external interrupt page 0
   ld      EIVR,#EIm_tlism+IntVect
   ; TLIS = 1, service routine vectors start at 10h
   ld      EIMR,#EIm_ia0m        ; maskable int. : INTA0 enabled
                                 ; all others disabled
   ld      EIPLR,#00000001B      ; priority levels
   ;        ++------------  INTA0 + INTA1 : 010 = 2
   ld      CICR,#Im_ienm+7  ; Interrupts enabled, program level = 7
   ret
; Interrupt service routine for WDT. Uses group 1 for working registers.
; The WATCHDOG Interrupt is used to wake up the ST9+ when
; the WFI instruction occurs.
;*** INTCLK = 125Khz
ReloadWDT:
   pushw   RPP              ; save current working register group
   srp     #2               ; select group 1
   ld      r1,PPR           ; save current page pointer
   spp     #RCCU_PG         ; select RCCU page
   or      CLK_FLAG,#Cm_xtdiv16 ;
;*** INTCLK = 2Mhz PLL is yet OFF
   ld      PLLCONF,#Cm_mul6+Cm_div1 ; PLL ON
   ld      r2,#42
Loop:djnz  r2,Loop          ; Wait 500µs for the PLL to lock
   or      CLK_FLAG,#Cm_csucksel ; INTCLK = PLL_CLOCK
;*** INTCLK = 12Mhz
   ld      PPR,r1           ; restore old PPR
   popw    RPP              ; restore working register group
```

```
    iret                    ; end of interrupt
```

3.6.2 Reset and Stop Manager

RESET normally means restarting from the beginning with everything initialized. However sometimes it's necessary to know the context of the ST9+ before the RESET and if it was an external or internal RESET. Two bits, SOFTRES and WDGRES in the CLK_FLAG register indicate the previous context. Table 3. shows the three cases to manage:

Table 3. Three Types of RESET

RESET Type	SOFTRES bit	WDGRES bit	Meaning
External RESET	0	0	High to low level on RESET pin.
Watchdog RESET	0	1	The Watchdog Timer is activated and the Timer has reached 0.
Software RESET	1	0	The HALT instruction is executed, waiting for an external Reset to restart (if the SRESEN bit in the CLKCTL register is set).

These bits are read-only and change with each Reset.

4 USING THE ON-CHIP PERIPHERALS

This chapter introduces the main peripherals of the ST9+ family. Each variant includes none, one or several peripherals of each type. This allows you to select the variant that best fits your requirements. For high-volume markets, you can order custom versions with exactly the type and number of peripherals required, including the relatively exotic ones not described here but available on request, such as videotext decoders etc.

4.1 PROGRAMMING THE CORE AND PERIPHERALS

In addition to describing how each peripheral works, we give examples of the code needed to use them in several configurations. This code is available on the Companion software. You can experiment with the original code or modify it to do the functions you require.

Configuring and using the core and the peripherals involves a considerable amount of bit manipulation in the registers. Many variables that drive microcontroller states are boolean values. To reduce the use of addressable space in registers, bits have been logically grouped in bytes, so the majority of the control registers have their eight bits fully used.

To properly program these registers, you need to track the exact position of each bit in each register. You can do this by referring systematically to the appropriate Data Sheet, and commenting the source text so that it can be easily read and understood later. See the following example:

```
    ld      R235,#11100000B ; R235 is register MODER
            ;      |||||||+-- HIMP : no foreign access to bus
            ;      ||||||+--- BRQEN : no foreign access to bus
```

```
        ;     |||+++---- PRS2,1,0 : processor at full speed
        ;     ||+------- DIV2 : crystal frequency divided by 2
        ;     |+-------- USP : user stack pointer in registers
        ;     +--------- SSP : system stack pointer in registers

spp     #0             ; SPI page
ld      R254,#10000001
        ;     |||||||+-- SPR0 : SPR1, SPR0 = 01 : clock divided by 16
        ;     ||||||+--- SPR1 :
        ;     |||||+---- CPHA : Input sampled on rising edge
        ;     ||||+----- CPOL : Rest level of serial clock = 0
        ;     |||+------ BUSY : Set to ready
        ;     ||+------- ARB : No I2C arbitration
        ;     |+-------- BMS : SPI used as a shift register (not I2C)
        ;     +--------- SPEN : SPI enabled
```

This style has the advantage of clarity. However, there is still a problem that it does not address: the variety of the different products of the ST9+ family.

The different products available in the ST9+ family are very diverse and as a result it is not always the case across products that a given function is performed by the same bit of the same register. The location of a register in one peripheral may be different in another. To avoid this problem, the GNU9 programming tool chain provides a set of include files that define the physical location of every bit and register, by their symbolic name. These files correspond to the appropriate variant of the ST9+. Using them guarantees that switching to another member of the family will not need more than changing the Include statement at the beginning of the source text. The same example as above, using the predefined symbols reads as follows:

```
.include ”system.inc” ; System register

ld      MODER,#MOm_sspm+MOm_uspm+MOm_div2m ; Both stacks in
                                   ;registers, clock divided by 2
spp     #SPI_PG                    ;SPI page
ld      SPICR,#SPm_spen+SPm_SP_16  ; Enable SPI,1st clock
                                   ; configuration, Clock/16
```

This notation is more compact, and is independent of changes either between variants in the same family, or by changes made globally to the family in the future. This writing style is recommended for this reason.

4.2 PARALLEL I/O

The parallel input-outputs have a basically very straightforward functionality. Once initialised, they appear as a register that can be written or read. However in many cases, direct byte-wide input-output is not sufficient. Bit-oriented I/O is often what is used in microcontroller systems. A powerful feature of the ST9+ is that you can address the eight bits of each port individually. The ST9+, like most microcontrollers also provides the external pins of the other peripherals (timers, UARTs, etc.) by diverting some bits from the parallel I/O ports.

The ST9+ parallel I/O has an additional very flexible feature. You can independently configure each bit as:

– An input with two variants (TTL or CMOS levels)

– An output with also two variants (open-drain or push-pull)

– A bi-directional port with either a weak pull-up or an open-drain output side

– An alternate function output (that is, the output pin of an internal peripheral), with also either open-drain or push-pull output driver.

– Analog Input (see note)

Note: On the port that accommodates the inputs of the Analog to Digital Converter, there is a special feature. In all other peripherals that require an input, you only need to configure the corresponding pin as an input. However, when using the ADC you can put the input pins to any voltage level from ground to Vcc. This is normally badly handled by standard logic gates that dissipate considerable power when the voltage reaches the limit range. To avoid this, the port that provides the input pins of the ADC has a special Alternate Function mode. Unlike the other ports it is used for input. This mode disconnects the input buffer from the pin and shorts the buffer input to the ground. The output buffer is put in high-impedance mode. The pin is permanently connected to the input of the ADC, thus allowing its voltage to be read at any time.

To handle all these capabilities, each port requires three configuration registers, PxC1, PxC2 and PxC3, where x is the port number. Once configured, a port exchanges data with the core through the PxDR data register.
The configuration registers are placed in various pages of the register group 15, as well as the data registers, except for the first six ports. These six belong to the system register group (group 14) for easy access.
Some ports also include DMA capability, configurable to work with the Multi-Function Timer.

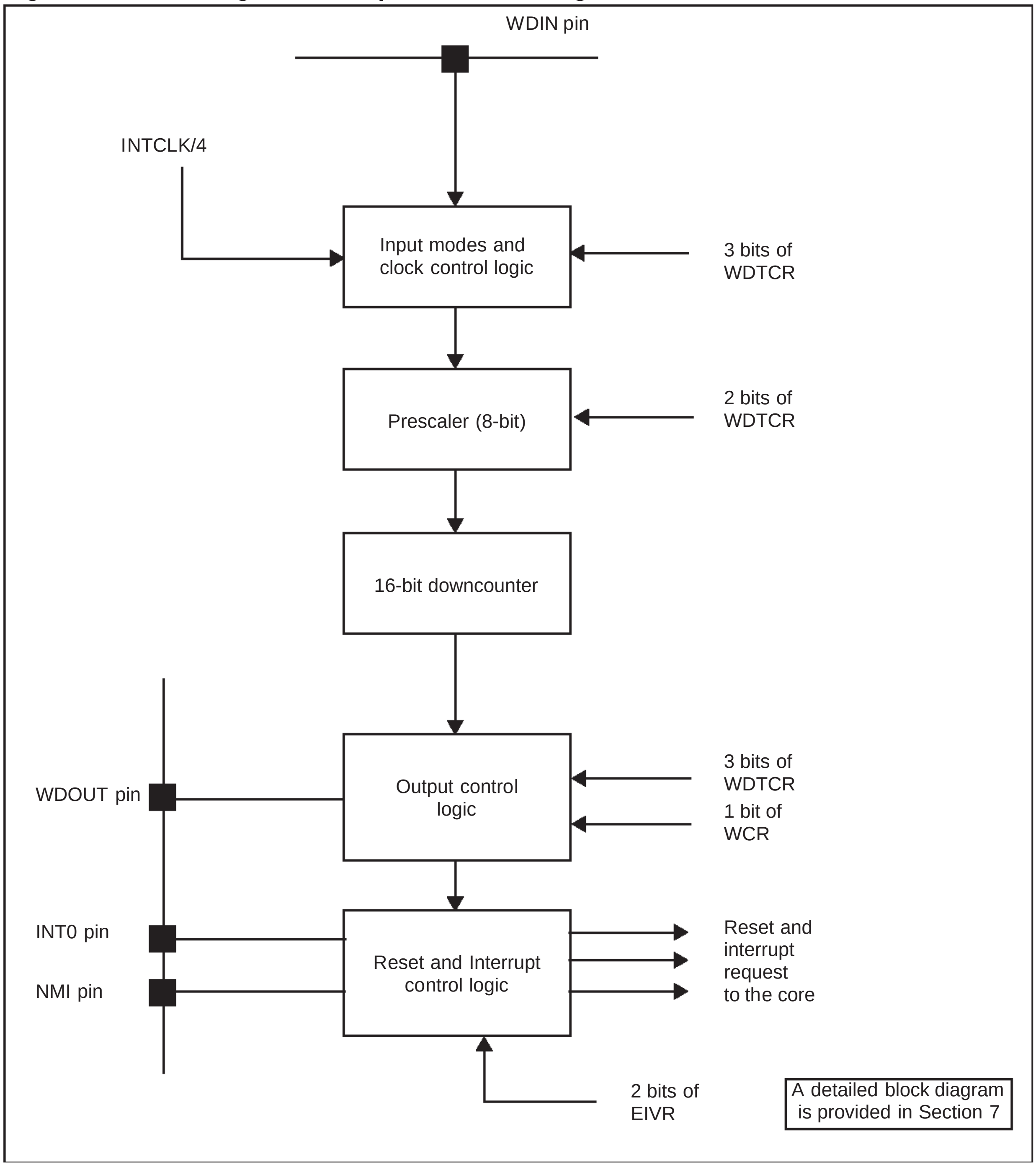
4.3 TIMER

You can use this timer both as a regular timer and as a watchdog timer.

4.3.1 Description

The block diagram of the Watchdog/Timer is the following:

Figure 24. Watchdog Timer Simplified Block Diagram



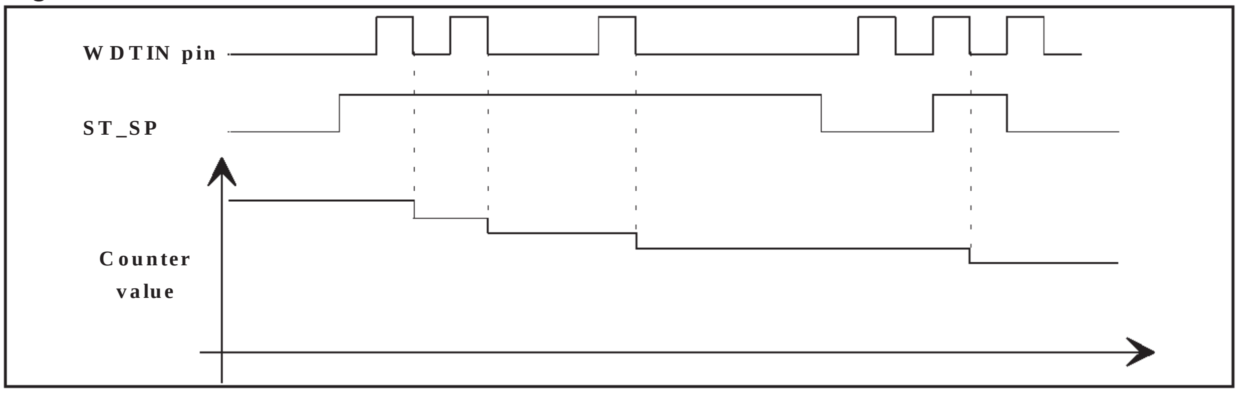
In counter/timer mode, the WDT can count pulses coming either from an input pin (WDIN; in the ST90158, alternate function of P7.0) or from the internal clock divided by 4. When the internal clock is used, the external pin, if it is enabled, can either gate the clock, start the counter, or reload it with its initial value. You select these modes using three bits in the WDT Control Register, as follows:

INEN	INMD1	INMD2	Mode
1	0	0	Event counter mode
1	0	1	Gated mode
1	1	1	Retriggerable input mode
1	1	0	Triggerable input mode
0	X	X	Input section disabled. Internal clock selected.

4.3.1.1 Event Counter Mode

The counter value is decremented at each falling edge on WDTIN pin if ST_SP is high (bit 7 of WDTCR).

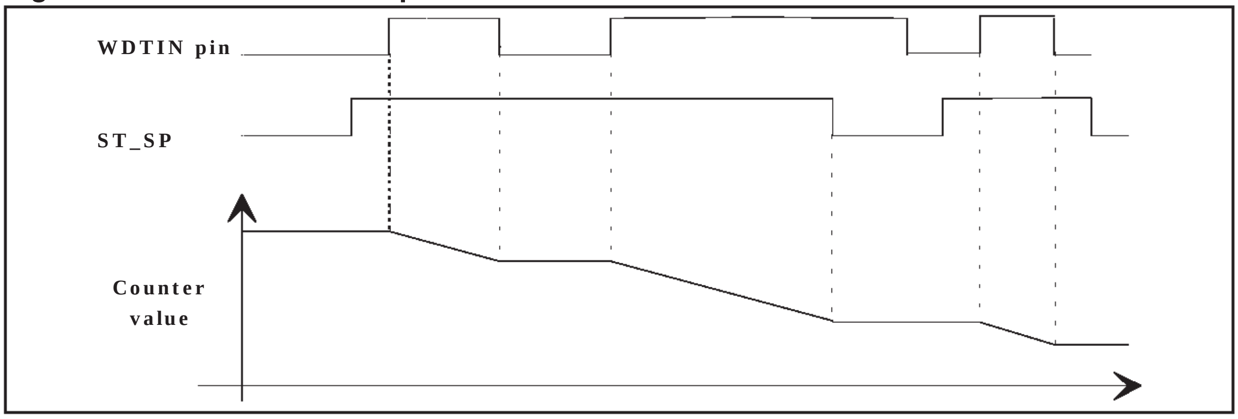
Figure 25. Timer in Event Counter Mode



4.3.1.2 Gated Input Mode

The counter value is decremented by WDTCLK (INTCLK / 4) if WDTIN pin and ST_SP are high.

Figure 26. Timer in Gated Input Mode

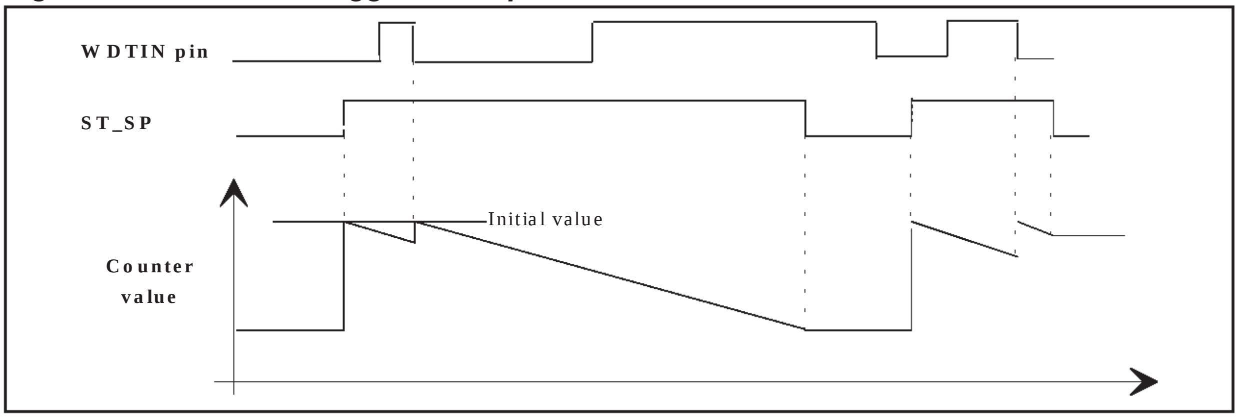


4.3.1.3 Retriggerable Input Mode

The counter value is decremented by WDTCLK (INTCLK / 4) if ST_SP is high.

The initial value is reloaded either at the rising edge of ST_SP or at each falling edge of WDTIN pin if ST_SP is high.

Figure 27. Timer in Retriggerable Input Mode

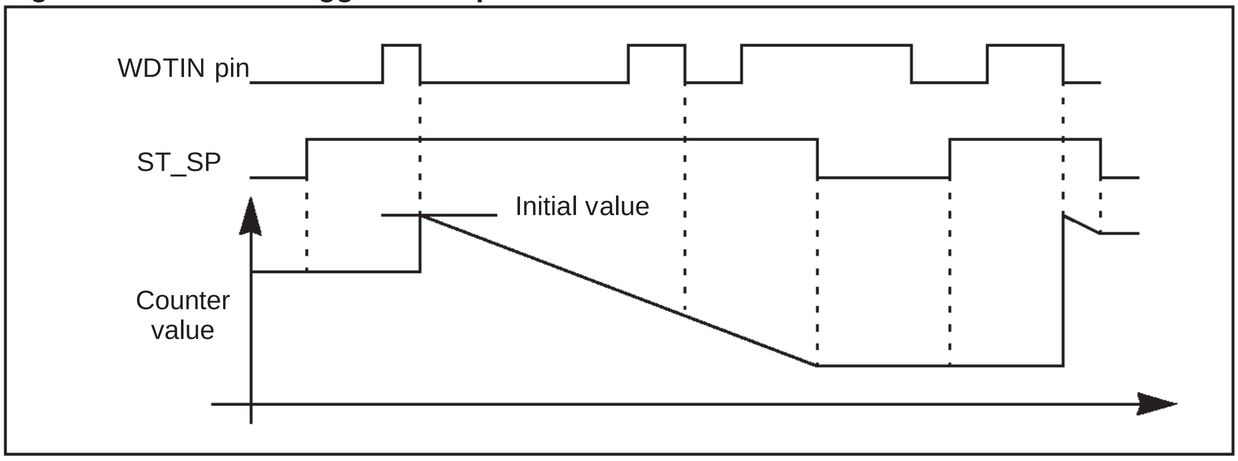


4.3.1.4 Triggerable Input Mode

The counter value is decremented by WDTCLK (INTCLK / 4) if ST_SP is high and falling edge of WDTIN occurs.

The initial value is reloaded at the first falling edge of WDTIN pin if ST_SP is high.

Figure 28. Timer in Triggerable Input Mode



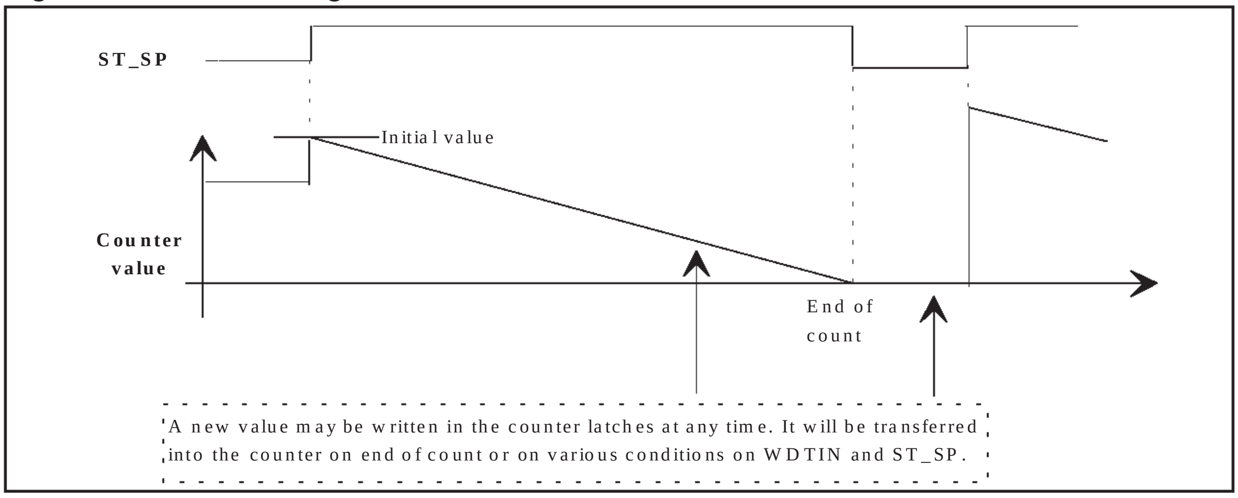
4.3.1.5 Single/Continuous Mode

On counter underflow (End Of Count), the counter is always reloaded with the value of the latch that is actually accessed when writing to the WDTHR and WDTLR register pair.

The counter has two modes: single shot or continuous, selected by the S_C bit of WDTCR. In single shot mode, the End Of Count also resets the ST_SP bit of the WDTCR, which stops the counter after one cycle. In continuous mode on reaching the End Of Count condition, the counter reloads the constant and restarts. When the ST_SP bit is set, the contents of the latch are written again to the counter, allowing the initial count value to be changed before starting the counter.

You restart the down counter by setting the ST_SP bit. The constant value can be either the initial value or a new one as shown on the following diagram:

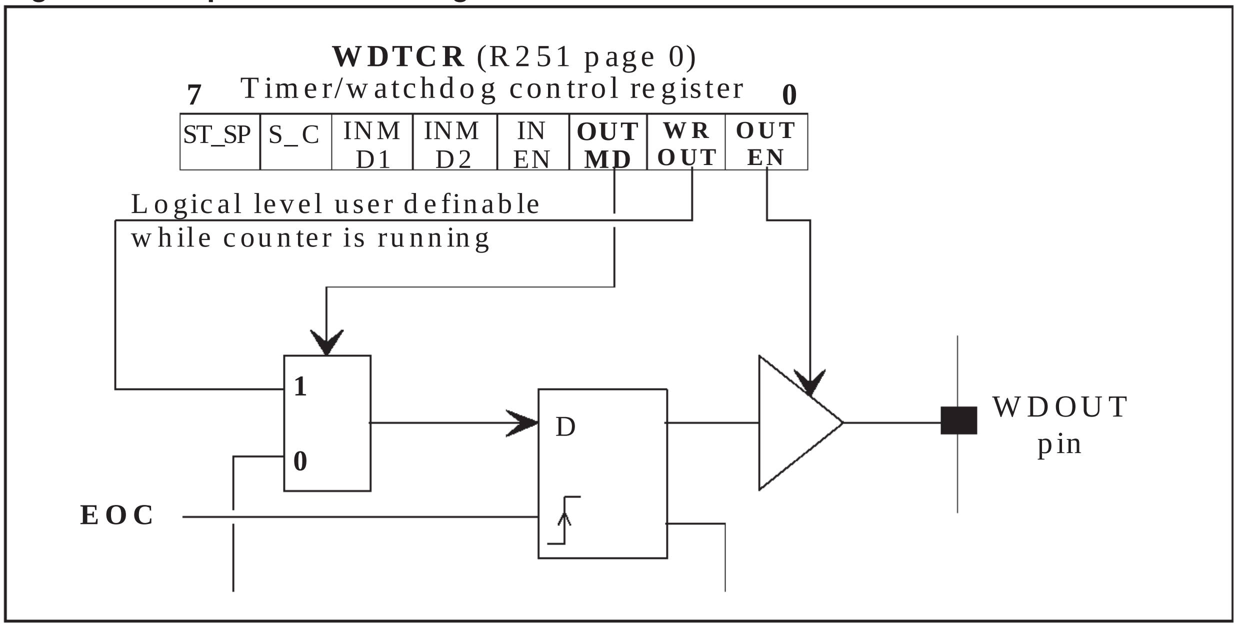
Figure 29. Timer in Single Mode


ST_SP
Initial value
Counter value
End of count
A new value may be written in the counter latches at any time. It will be transferred into the counter on end of count or on various conditions on WDTIN and ST_SP.

4.3.1.6 Output pin

Another pin, WDOUT (alternate function of P8.4 for ST90158), when enabled by the OUTEN bit, can change its state in two ways on the end of count of the main counter. Basically, each time the counter overflows, it updates the output value. This can produce two different effects, selected by the OUTMD bit: either the state of the WROUT bit is copied to the output at that time, or the output is complemented.

Figure 30. Output Pin Block Diagram



4.3.2 Timer Application for Periodic Interrupts

In this application, the clock is internal and the input and output pins are unused. The counter is set to continuous mode, and the value of the reload registers chosen so that the overflow occurs exactly every 128 microseconds.

4.3.2.1 Initialisation of the WDT for a Periodic Interrupt

Very few registers are involved when you use the WDT for this purpose, since there is no input apart from the internal clock and no output. However, the WDT interrupt handling is a little different from most other peripherals in that it borrows the interrupt circuit named INTA0 that is normally assigned to the external interrupt pin INT0. So you configure the WDT in two steps: initialising the WDT itself, and also the external interrupt INTA0.

In this example application, the crystal frequency is 24 MHz and the interrupt rate must be 8192 Hz. The timer starts with the preset count on the low-to-high transition of the ST_SP bit of the WDT Control Register.

The initialisation routine for the WDT is then:

```
/* *********** configure watchdog for periodic interrupt *************/
void ConfigWDT ( void )
   {
   SelectPage( WDT_PG )      ;/* select Watchdog page */
   WDTPR = 0 ;/* 122 us = 366 ticks */
   WDTR = 365 ;/* preset is nb ticks-1 (WDTR is the pair WDTLR and WDTHR) */
   WDTCR = (WDTm_stsp) ;/* WDT is set to continuous mode, no inputs, no
   outputs */
   /* Interrupt must be connected to intA0 in the EIVR register */
   }
```

The initialisation routine for INTA0 follows. In this example application, the interrupt A0 is the only ”external” interrupt enabled, and it is assigned priority level 2:

```
/* *********** initialise interrupt A0 and current level **************/
void ConfigInterrupts ( void )
   {
   SelectPage( EXINT_PG ) ;
   EIVR = (EIm_tlism+(unsigned char)INTA0VECT) ;/* int. A0 is generated
   on WDT overflow */
   EIPLR = 1 ;/* intr A0 (and that of same group) with high priority */
   EIMR = EIm_ia0m ;/* intr A0 alone enabled */
   CICR = (Im_gcenm+Im_iamm+Im_ienm)+7 ;
   /* current processing level : minimum ; int. enabled, nested mode */
   /* This starts also the MFT */
   }
```

4.3.3 ST90158 Rectangular Signal Generator Application (PWM)

This application provides a pulse of 0.1s every second at the output pin of the WDT. You can use it as an application exercise for learning the programming and debugging tools, since it is very simple. The only hardware required to watch the effect of the program is a LED in series with a resistor connected between P8.4 and Vcc (anode towards Vcc). You will find it in the Companion software in the directory \programm\periph\wdtrect.*.

The WDT can only provide a delay after which the output may change its state, and an interrupt is triggered. The principle of using the WDT as a rectangular signal generator is to set it to the continuous mode, to load it with a time value, and let it count down until zero. The control register is set so that an interrupt is then generated, and the output pin is updated at the same time. The interrupt service routine will reload with the other value, and preset the WROUT bit to the complement of the current value, so that the opposite state will be transferred to the output pin at the next end of count. By alternating between two time values, a duty cycle other than 50% can be obtained (PWM). When the interrupt occurs, the output has already changed its state, so that the waveform can be very precise since it depends only on the timer hardware and not on the software. All that the interrupt service routine has to do is to load the timer latch with the value to be used when the end of count is reached. Thus, the constraint is that the reload interrupt must be guaranteed a latency time less that the shortest time between two output transitions. This may or may not be difficult to realise according to the intended timing and the presence of portions of the program where the interrupts are disabled. It is thus recommended to properly manage the interrupts and thoroughly use the priorities to achieve this requirement.

The WDT does not have an interrupt cause and a vector of its own. It must borrow them either from the Top Level Interrupt or the INTA0 input. We have chosen to use INTA0 here, and to give it a priority of 2 which is high, but not the highest (which is 0).

4.3.3.1 Creating the Template start-up file

The software package provides all the start-up programs you will need to start your applications.

The gmake executable (see the GNU Make utility user manual) will automatically generate the ”crt9s.asm” start-up file with the command below (for the ST90158 makefile of the gnu9p\samples\st90158 directory):

gmake conf1

or

gmake conf2

This command will convert the ”crt9s.c” file from the directory bin\src\cstart to ”crt9s.asm” depending on the conf1 or conf2 option which initializes the four data segments (DPR0 to DPR3).

To use the ”crt9s.asm” for our application you must add the call to the WDT interrupt service routine by changing the following instructions:

```
   .text
   .word  __Reset
   .blkb   50               ; reserve room for the interrupt vectors
```

by

```
   IntVect = 10h            ; origin of interrupt vector table
   .global IntVect          ; to use it in the initialisation Interrupt
                            ; System
   .extern ReloadWDT        ; address of the interrupt service routine
   .text
   .word   __Reset
   .org    IntVect
   .word   ReloadWDT        ; INTA0 interrupt vector
   .blkb   50               ; reserve room for the interrupt vectors
```

At the end of this program, a call to the main program is done. This main program is a C file which is in our application ”sample1.c” with the following instructions:

```
extern void InitWDT_Rect();
extern void InitIntr();
void main(void)
{
   int j;
   InitWDT_Rect();
   InitIntr();
   for (j=0;;j++)     /* loop for ever      */
   {}
}
```

4.3.3.2 Initialising the WDT for a Rectangular Signal Generator

This piece of code initialises both the WDT and port 8 to use P8.4 in alternate function. It selects INTA0 as the associated interrupt.

```
InitWDT_Rect:
   spp     #WDT_PG          ; select watchdog page
   ld      WDTPR,#127       ; prescaler 1/128
   ldw     WDTR,#LONG       ; start with long time
   ld      WDTCR,#WDm_stsp+WDm_outmd+WDm_wrout+WDm_outen
   ; output enabled, WDOUT pin level at end of count
   ; output pin mode, input disabled, clock = INTCLK/4
   ; continuous mode, start operation
   spp      #P8C_PG         ; Port 8 control registers page
   ld      P8C0R,#10h       ; Configure P8.4 as alternate function,
                            ; push-pull
   ld      P8C1R,#10h       ; Other pins bidirectional.
   ld      P8C2R,#0
   ret
```

4.3.3.3 Initialising the Interrupt System

This piece of code selects the vector address and the interrupt level of INTA0. It also enables the interrupts globally.

```
InitIntr:
   spp     #EXINT_PG            ; external interrupt page 0
   ld      EIVR #EIm_tlism+IntVect; IntVect less than 0F0h
   ; TLIS = 1, service routine vectors start at 10h
   ld      EIMR,#EIm_ia0m       ; maskable int. : INTA0 enabled
                                ; all others disabled
   ld      EIPLR,#00000001B     ; priority levels
   ;                    ++------  INTA0 + INTA1 : 010 = 2
   ld      CICR,#Im_ienm+7  ; Interrupts enabled, program level = 7
   ret
```

4.3.3.4 Interrupt Service Routine

This interrupt service routine reloads the WDTR with one of two values to alternately provide a short and a long delay. The high output level (LED off) is associated to the long delay and, the low level (LED on) to the short one. At the beginning, the current output level as defined by the WROUT bit (WDTCR.1) is checked to define whether the previous time was a long or a short one. The reload value is chosen in consequence.

```
; Interrupt service routine for WDT. Uses group 1 for working registers.
; with a INTCLK/4 = 3 MHz and a WDTCLK divided by 128, the values to be
; loaded in WDTR are :

SHORT = 2344; on time = 0.1 s  (low level on WDTOUT pin)
LONG = 21094; off time = 0.9 s  (high level on WDTOUT pin)

ReloadWDT:
   pushw   RPP              ; save current working register group
   srp     #2               ; select group 1
   ld      r1,PPR           ; save current page pointer
   spp     #WDT_PG          ; select watchdog page
   ld      r0,WDTCR         ; get control register
   btjt    r0.1,LoadShort   ; if 1, was long time

   ; was short time : preload for long
   ldw      WDTR,#LONG       ; start with long time
   or      WDTCR,#2          ; set output value high for next time
   ld      PPR,r1            ; restore old PPR
   popw    RPP               ; restore working register group
   iret                      ; end of interrupt

LoadShort:
   ldw     WDTR,#SHORT       ; start with short time
   and     WDTCR,#~WDm_wrout ; clear output value low for next time
   ld      PPR,r1            ; restore old PPR
   popw    RPP               ; restore working register group
   iret                      ; end of interrupt
```

4.3.4 Watchdog Application

A watchdog timer is a safety measure to prevent a program going adrift. It relies on a hardware timer that must be periodically reset by the program. Failing to do this will reset the whole program (at the End Of Count).

The efficiency of this approach varies with the type of program and depends on the following conditions:

The hardware action to perform in order to reset the timer must be so complex that if the processor goes adrift, it cannot accidentally reset the watchdog. In the ST9+, you need to write 0AAh and 55h successively to the WDTLR.

The chosen time-out value must be greater than the interval at which the program resets the watchdog. It must also be as close as possible to that interval for maximum safety. Since the watchdog time-out value is set once at the beginning of the program, this condition is best fulfilled if the resetting action is performed at constant intervals.

A special software arrangement must be designed to reduce the chance that the part of the program that resets the watchdog can continue undisturbed when other parts of the program are faulty. This may include an interlock mechanism that requires that several program branches are executed to enable the resetting of the Watchdog count.

As you can see, achieving a very secure program malfunction detection by the sole means of a watchdog is a very difficult thing to realise. However, the watchdog can still play a key role in the safety of some systems, of which an example is an induction motor controller.

4.3.4.1 Initialising and Resetting the Watchdog

The watchdog is initialised with the following code:

```
; Initialisation of the watchdog timer.
```

```
InitWDT::
   spp     #WDT_PG
   ldw     WDTR, #900        ; 300 us at 12 MHz internal
   ld      WDTPR, #0         ; division by 1
   ld      WCR, #0           ; Start Watchdog operation
   ret
```

You start it by resetting the WDGEN bit of the WCR register (R252 page 0). You do this at the time all the initialisations are performed and the program effectively starts, otherwise the watchdog could ”bite” too early.

Executing the following code does the reset:

```
   spp     #WDT_PG           ; Restart Watchdog
   ld      WDTLR, #0AAh      ; This sequence is conventional
                             ; so that there is little chance that
   ld      WDTLR, #55h       ; it be produced by mistake
```

4.4 SERIAL PERIPHERAL INTERFACE

4.4.1 Description

The SPI is a synchronous input-output port that you can configure in various modes, including S-Bus. It can have many more uses, of which two are considered here: interfacing with serial-access EEPROMs, and interfacing with a liquid-crystal display.

The main block of the SPI is an 8-bit shift register which can be read or written in parallel through the internal data bus of the ST9+, and that can shift the data in or out on two separate pins, named SDI and SDO, respectively. The serial transfer is initiated with a write to the SPI Data Register (SPIDR). Input and output are done simultaneously, each most significant bit being output on SDO while the level at SDI becomes the least significant bit. Each time a bit is transferred, a pulse is output at the SCK pin. When eight bits are transferred, eight pulses have been sent on SCK, and the process stops. If the proper bits are set in the SPI Control Register (SPICR), an interrupt can be requested on end of transmission. To summarise:

Transfers are started by writing a byte into the SPI data register

Input and output are done at the same time

Input and output are done most-significant bit first

Eight clock pulses are output on the SCK pin synchronously with bit shifting

An interrupt request can be issued on end of transmission

The SPI is configured with the SPICR register. Four bits of this register are of special interest in the applications detailed here: these are the CPOL-CPHA pair, and the SPR1-SPR0 pair.

The CPOL-CPHA pair defines the polarity and phase of the clock pulses. This allows them to adapt to the external device. CPOL selects between rising edge or falling edge as the active transition, and CPHA selects whether the first active edge is the first edge of the pulse train or the second one.

The SPR1-SPR0 pair selects one of four different transfer speeds.

4.4.2 Static Liquid-Crystal Display Interface Example

This example uses a static liquid-crystal display that shows a simple number. This can be the voltage in a digital voltmeter, or any 3 1/2 digit value. A very simple demonstration program is supplied in the Companion Software. Some of the routines are reused in the bar code reader described later. The path is \programm\periph\spilcd.*.

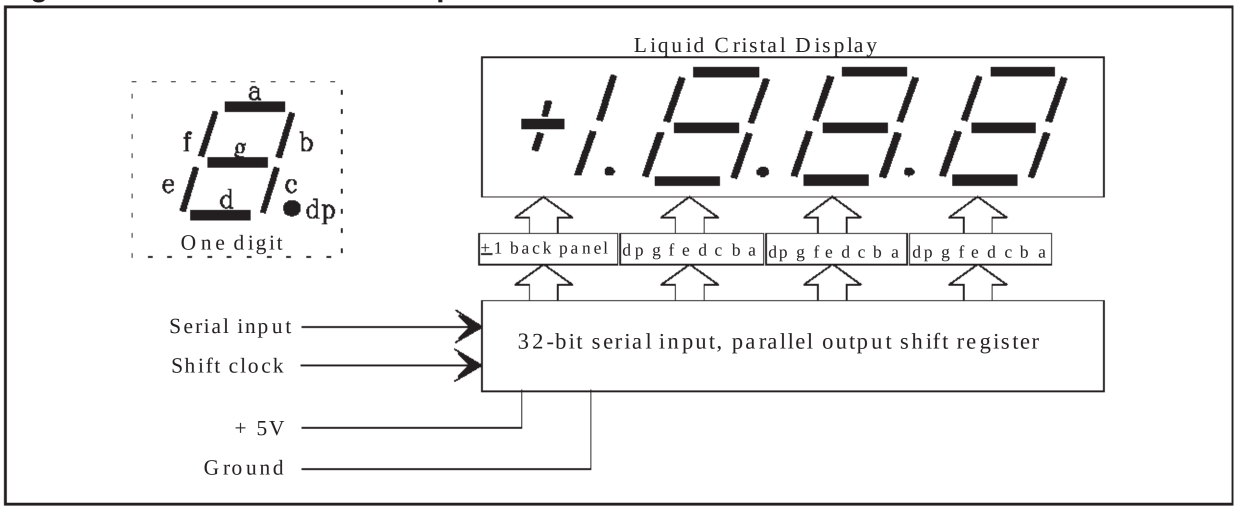
A static liquid-crystal display is composed of a glass back-panel, and a set of front-panel electrodes printed on a glass front-panel. The space between both panels is filled with a liquid-crystal solution. The electrodes are transparent, and the panels may be fitted with a combination of polarisers, so that the whole display, unpowered, is either transparent or opaque according to the selected polariser combination. When a difference of potential is applied between one electrode and the backplane, the area delimited on the front panel by the shape of the electrode takes the reverse state, i.e. opaque or transparent, respectively. Using a seven-segment pattern, it is possible to display numbers, by activating the appropriate segments to show the desired figures.

A liquid-crystal display presents a high impedance between its electrodes, ranging typically in the megaohm region. It is thus a voltage-controlled device, with a threshold of about 3 V to change from one state to the other. Driving such a display is then a simple matter, but one fundamental precaution must be taken to avoid premature destruction of the display: it must be driven with alternating voltage, to avoid electrolysis within the inter-electrode solution. The frequency is of little importance, and the best values range between 30 and 100 Hz.

This basic explanation does not cover all the details and precautions relative to LCD’s but is sufficient to understand the use of the SPI as a LCD driver.

In this application, the display is installed on a separate board. Since it has 40 pins, it is not convenient to connect this board with a 40-wire cable. Using a serial synchronous transmission, only four wires are required. The block diagram of the circuit of the display board is:

Figure 31. LCD Interface Example



As shown by this diagram, to make it easy to wire our display board, the correspondence between the segment positions and the bit positions in the bytes transferred is actually:

Bit position	7	6	5	4	3	2	1	0
Segment	dp	g	f	a	b	c	d	e

And the conversion between the figures (0 to 9) and the display pattern is done by a constant array defined in this case as:

```
; Table of characters. Gives the patterns for the last 3 digits, from 0 to 9.
Characters:
   .byte 3Fh, 0Ch, 5Bh, 5Eh, 64h, 76h, 77h, 1Ch, 7Fh, 7Eh
```

The LCD electrodes are connected to the outputs of a serial input, parallel output shift register. The backplane is also connected to one output. This allows us to provide the A.C. drive for the electrodes, as shown in the table below:

Level of the backplane	Level of one of the segments	State of the liquid crystal between the electrodes
0	0	off
0	1	on, positive voltage
1	0	on, negative voltage
1	1	off

In short, the segment is On if its level is different from that of the backplane. It is Off if its level is the same as that of the backplane. Thus, to generate the drive pattern, it is only necessary to build the four bytes that correspond to the various segments, keeping the bit corresponding to the backplane at level zero. Then, using a periodic interrupt, these four words are transmitted serially, but complemented every other time. As an example, to display the value 375, as in the demonstration program, the following message is built:

Symbols	Hundreds	Tens	Units
00000000	01011110	00011100	01110110

Since in this original message the backplane bit is kept at zero, all bits that are 1 correspond to an activated segment; the others remain invisible. This message is sent at intervals of about 16 ms, but it is inverted every other time, giving the succession:

Symbols	Hundreds	Tens	Units
00000000	01011110	00011100	01110110
11111111	10100001	11100011	10001001
00000000	01011110	00011100	01110110
11111111	10100001	11100011	10001001
00000000	01011110	00011100	01110110
11111111	10100001	11100011	10001001
etc.			

This succession of inverted words guarantee a zero D.C. component across the liquid crystal solution, which is absolutely necessary.

4.4.2.1 Initialising the SPI

This routine is very simple since there is only one configuration register in the SPI. Of course, since a periodic interrupt is used to trigger the transmission routine, the WDT timer is also initialised, as well as the external interrupt unit, much in the way described in Section 4.3.2.1 that discusses the WDT, but for a interrupt rate of 60 per second.

No other peripheral is required except the parallel port 9 that is configured to allow the use of P9.6 and P9.7 as alternate functions. The initialisation code of port 9 reads as follows

You can get the complete source code to experiment with from the comp (in the files named SPILCD.*):

```
   spp      #P9C_PG                 ; Port 9 control registers page
   ld       P9C0R,#11000000B
   ld       P9C1R,#11000000B        ; pins P9.6, P9.7
                                    ; alternate function SCK, SDO
   ld       P9C2R,#00000000B
```

The initialisation code for the SPI is the following:

```
; Initialisation of the SPI as a LCD output
   .text
InitLCDthruSPI:
   spp      #SPI_PG                 ; SPI page
   ld       SPICR, #SPm_spen        ; Enable SPI, 1st clock configuration,
                                    ; top speed
   ld       LCDPolarity, #0         ; Clear phase variable
   ret
```

4.4.2.2 Transmission Routine

The transmission is triggered by the WDT interrupt at a rate of 60 times per second. It uses a variable LCDPolarity that is toggled at each interrupt between values 0 and 0FFh. The four bytes corresponding to the 31 segments plus the backplane are XORed with the LCDPolarity value, so that the bytes are reversed every other time. Before sending each byte, the busy status is checked and the routine waits until it is false, i.e. until the transmit buffer is free.

```
; Interrupt service routine of SPI to refresh LCD
; ***********************************************
; the pattern to display is stored in data memory as follows:
   .bss
LCDPattern:
   ds       4                 ; low-order first
```

```
LCDPolarity:
   ds      1                ; toggles between 0 and 0FFh
; Interrupt service routine. Uses register group 1.
;
   .text
RefreshLCD:
   pushw   RPP              ; save current working register group
   srp     #2               ; select group 1
   ld      r0, PPR          ; save current page pointer
   spp     #SPI_PG          ; SPI page

   ldw     rr4, #LCDPattern ; address of storage of pattern
   ld      r3, LCDPolarity  ; current polarity.
   cpl     r3
   ld      LCDPolarity, r3  ; store reversed polarity
   ld      r2, #4           ; count of bytes to send

WaitSPPfree:
   ld      r6, SPICR        ; test if SPI busy
   btjt    r6.4, WaitSPPfree ; wait until ready

   ld      r1, (rr4)+       ; get first byte
   xor     r1, r3           ; complement or not according to current
                            ; polarity
   ld      SPIDR, r1        ; send byte

   djnz    r2, WaitSPPfree  ; count exhausted ?
   ld      PPR, r0          ; restore old PPR
   popw    RPP              ; restore working register group
   iret                     ; end of interrupt
```

4.4.3 EEPROM Serial Interface Example using I^{2}C

An I^{2}C Serial EEPROM is a very convenient device that stores a few bytes (up to 1024) for at least 10 years in a secure manner. The serial I^{2}C access uses only two wires, which allows the memory to fit in an 8-pin DIL package. The serial access requires a serial transmission protocol, which makes the access more difficult than with a parallel-bus EEPROM device. This, however, is not a real drawback, for two reasons:

– The serial synchronous transmission is very easy to implement, and is not time-critical

– Using a serial protocol makes it impossible (or very unlikely) that you will overwrite or erase data by mistake, which ensures a high safety level for the application

Thus, for applications that can manage with a small amount of permanent data, a serial EEPROM is the best choice. Examples include storing the reading of a counter or a recording meter, storing configuration parameters or calibration values. The application described here uses a PCF8594E memory chip, that takes advantage of the SPI to exchange data with minimum software overhead.

4.4.3.1 Additional clock timing

Compared to the I^{2}C timing SPI timing needs additional software to manage:

– The start condition

– The ninth bit for acknowledgement

– The stop condition

These three timing differences are implemented by changing the input/output port data and configuration registers, to pull the level on the SDA and SCL pins up or down.

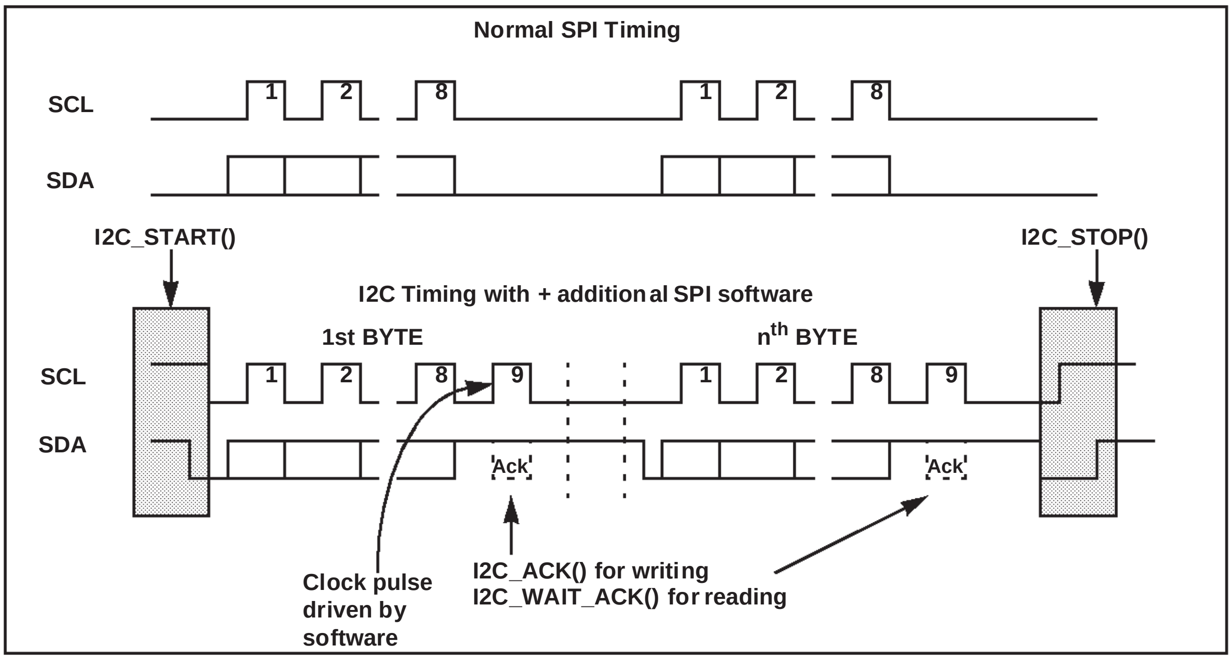
In the example the two acknowledge management sub-routines are ”I2C_ACK()” for the output data and ”I2C_WAIT_ACK()” for the input data.

The start condition sub-routine is ”I2C_START()”, and the stop condition sub-routine is ”I2C_STOP()”.

Figure 32 shows the I^{2}C Timing implemented with additional SPI timing software.

All of these sub-routines are in the ”I2C.C” file.

Figure 32. I²C Data and Clock Timing with Additional SPI Software



4.4.3.2 EEPROM I²C protocol

To connect more than one device to an I²C interface, each device needs an address to be selected. The EEPROM PCF8594E used for our example has an address equal to A0h.

Following a start condition the bus master (SPI) must output the address of the EEPROM it is accessing. The most significant four bits are the device type identifier (1010). The next bit (bit 1) is the page selection bit and the last bit (bit 0) defines the operation to be performed for the next byte to transfer (1 for read, 0 for write). After the address has been sent, seven data bytes can be sent to the EEPROM for writing, and all the data bytes can be read successively from the EEPROM for verification.

4.4.3.3 Sub-routine for I2C Interface

I2C.C file:

```
#include <io_port.h>
#include ”i2c.h”
#include ”define.h”
/**-*-*-*-*-*-*-* LIBRARY FUNCTIONS -*-*-*-*-*-*-*-*-*-*-*-*-*-*-*/
/*-------------------------------------------------------------
ROUTINE Name : INIT_I2C()
Description :  Port initialization :(ST90158!)
                     P9.6 : SCL (I2C Clock)
                     P9.7 : SDO (I2C Data)
   These two pins are initialized as :ALTERNATE FUNCTION, OPEN DRAIN, TTL
   The other pins of port9 are : BIDIRECTIONAL, WEAK PULL UP, TTL
                     P7.1 : SDI  (I2C Data)
   This pin is initialized in INPUT.
   The SPI is configured to work with I2C protocol
   For the moment, I2C bus remains disabled (clock not generated).
Comments : If INTCLK > 12.8 MHz, you must change the value of constant
   I2C_SPEED in i2c.h file.
-----------------------------------------------------------------*/

void INIT_I2C (void){
   spp(P9C_PG);
   P9C0R = 0xC0;
   P9C1R = 0xC0;
   P9C2R = 0xC0;
   spp(P7C_PG);
   P7C0R = 0x02;
   P7C1R = 0x00;
   P7C2R = 0x02;
   spp(SPI_PG);
   SPICR = SPm_bms + I2C_SPEED;     /* I2C bus disabled : */
}
/*------------------------------------------------------------------
ROUTINE Name : I2C_WAIT()
Description :  It is a loop of 4.7µs (including call and ret cycles)
        srp.............4 cycles
        ld..............4 cycles
        djnz............6 cycles
        call+ret........22 cycles
        With an INTCLK of 12MHz, we need 57 cycles.
Comments :    This routine is written for INTCLK = 12MHz
------------------------------------------------------------------*/
void I2C_WAIT(void){
   asm("               ld   r0,#004h
          delay:     djnz   r0,delay
     ”);
}
/*-------------------------------------------------------------------
ROUTINE Name :  I2C_START()
Description :  It generates a start signal for I2C communication
Comments :  While SCL remains high, SDA turns from high to low level.
-----------------------------------------------------------------*/
void I2C_START(void){
   spp(SPI_PG);
   SPICR &= ~SPm_spen;   /*Disable I2C bus Compulsory because I2C_START()
                           can be used after I2C_WAIT_ACK() for read
                         operations*/
   spp(P9C_PG);
   P9DR = 0X40;       /*To force SDA to Low level from Internal Data bus,*/
   P9C0R = 0x40;     /*Output Mode is necessary*/
   I2C_WAIT();       /*4.7µs wait : SD remains low*/
   spp(SPI_PG);
   SPICR |= SPm_spen;   /*I2C bus enabled : 8 clock pulses will be
                          generated*/
   spp(P9C_PG);       /*SDA in Alternate Function ,Open Drain :*/
   P9C0R = 0xC0;        /*Thus, SDA turns to High level because of pull-up
                          resistors*/
}
/*-------------------------------------------------------------------
ROUTINE Name :  I2C_STOP()
Description :  It generates a stop signal for I2C communications
Comments :  While SCL remains high, SDA turns from low to high level.
-----------------------------------------------------------------*/
void I2C_STOP(void){
   spp(P9C_PG);
   P9DR = 0x40;       /*SDA is configured in Output Mode*/
   P9C0R = 0x40;      /*So it is possible to force it to low level*/
   P9C1R = 0xC0;
   P9C2R = 0xC0;      /*SCL remains in Alternate function, Open Drain*/
   spp(SPI_PG);       /*I2C bus disabled : SCL is in HIGH level*/
   SPICR &= ~SPm_spen;
   I2C_WAIT();        /*4.7µs wait */
   spp(P9C_PG);       /*SDA configured in Alternate function, Open Drain*/
   P9C0R = 0xC0;      /*So SDA turns to High level (pull-up resistors)*/
   P9C1R = 0xC0;
   P9C2R = 0xC0;      /*Finally, SDA has turned from low to high level while
                        SCL remained high*/
}

/*-----------------------------------------------------------------
ROUTINE Name : I2C_WAIT_ACK()
Description :  This function is aimed at waiting for an ACK from the slave.
        To achieve an ACK, the slave must force SDA to Low level.
        The ST9+ does a polling on SDA pin configured in input, during
   "timeout".
        If an ACK does not occur during this time, a STOP is generated, and
        the communication ends.
        If an ACK occurs before the end of timeout, the process leaves
        the function, ready to send new data.
Comments :
-----------------------------------------------------------------*/
void I2C_WAIT_ACK(void){
   unsigned int timeout;

   spp(SPI_PG);
   SPICR &= ~SPm_spen;   /*I2C disabled*/
   NOP;
   NOP;
   spp(P9C_PG);
   P9C0R = 0xC0;      /*SDA is configured in INPUT and remains high for the
                        moment*/
   P9C1R = 0x40;
   P9C2R = 0x40;
   timeout = 0x00300;   /*time when the ST9+ is waiting for the ACK from the
```

```
                    slave*/
   while(((P9DR & 0x80) != 0) && (timeout > 0))  /*expecting SDA to turn LOW
                                                  (forced by slave)*/
           timeout--;
   if(timeout == 0){    /*if no ACK from the slave*/
           I2C_STOP();  /*the ST9+ generates a STOP sequence and close the
                          communication*/
           return;
   }
                    /*from here, an ACK has been received*/
   I2C_WAIT();
   spp(SPI_PG);
   SPICR |= SPm_spen;   /*I2C bus is enabled*/
   spp(P9C_PG);
   P9C0R = 0xC0;
   P9C1R = 0xC0;      /*SDA configured in Alternate Function, Open Drain*/
   P9C2R = 0xC0;      /*SCL remains in Alternate Function, Open Drain*/
}
/*-------------------------------------------------------------------
ROUTINE Name :  I2C_ACK()
Description :   This function generates a software ACK from the ST9+ :
    During 4.7µs, SCL remains high while SDA is forced to low level.
Comments :      Used in read operations
------------------------------------------------------------------*/
void I2C_ACK(void){
   spp(P9C_PG);
   P9DR = 0x40;       /*SDA configured in Output, forced to LOW level*/
   P9C0R = 0x40;
   spp(SPI_PG);
   SPICR &= ~ SPm_spen;  /*I2C bus disabled : SCL turns from LOW to HIGH*/
   I2C_WAIT();        /*4.7µs wait*/
   spp(SPI_PG);
   SPICR |= SPm_spen;   /*I2C bus enabled : SCL turns from HIGH to LOW*/
   spp(P9C_PG);       /*SDA configured in Alternate Function ,Open Drain*/
   P9C0R = 0xC0;      /*So SDA turns to High level (pull-up resistors)*/
```

```
}
/*-----------------------------------------------------------------
ROUTINE Name :  I2C_SEND_DATA()
Input:      unsigned char sendbyte : the byte to send to the slave (data, I2C
            address...)
Description :  sendbyte is sent on the I2C bus
Comments :  - This function can be used AFTER : I2C_START() or
            I2C_WAIT_ACK()
            - The process remains in the fonction while Busy Flag remains in
             HIGH level(= the transmission is not finished)
   If your software uses interrupts to send data on I2C bus, delete the last
   line of this function (polling).
-----------------------------------------------------------------*/
void I2C_SEND_DATA(unsigned char sendbyte){
   spp(SPI_PG);
   SPIDR = sendbyte;   /*data to send is loaded into data register*/
          /*I2C bus has been enabled in I2C_START() or in I2C_WAIT_ACK ()*/

   while ((SPICR & SPm_sp_busy) == SPm_sp_busy);  /*Polling on busy flag*/
}
/*-----------------------------------------------------------------
ROUTINE Name :  I2C_READ_DATA()
Description :  0xFF is written in data register to start the transmission
Comments :  - This function must be followed by I2C_ACK()
            - If your software doesn’t use interrupts to read data from I2C
            bus :
   * Check that the busy flag is cleared before leaving this function.
   * Modify this function so that it returns the read value
   ( unsigned char I2C_READ_DATA(void) )
-----------------------------------------------------------------*/
void I2C_READ_DATA(void){
     spp(SPI_PG);
     SPIDR = 0xFF;      /*0xFF is loaded into data register*/
                /*The slave will replace wrong ”1” with ”0”*/
          /*I2C bus has been enabled in I2C_START() or in I2C_WAIT_ACK ()*/
```

```
}
/*-*-*-*-*-*-*-*-*-*-*-* INTERRUPT FUNCTIONS -*-*-*-*-*-*-*-*-*-*-*-*/
/*-----------------------------------------------------------------
ROUTINE Name :  INIT_I2C_IT
Description :This routine configures the interrupts after each I2C End of
           Transmission using interrupt channel INTB0.
Comments :    Don't forget to initialize your startup file correctly.
------------------------------------------------------------------*/
void INIT_I2C_IT(){
   di();               /* Disable IT before modifying EIVR register*/
   spp(EXINT_PG);
   NOP;                /* compulsory*/
   EIVR = (IT_EXT_VECT & 0xF0);
   EIPLR |= (PRIO_I2C >> 1)*4;/* Set the priority*/
   ei();               /* Enable IT */
}

/*-----------------------------------------------------------------
ROUTINE Name : Enable_I2C_IT
Description :  Enable I2C interrupts on channel INTB0
Comments :
-----------------------------------------------------------------*/
void Enable_I2C_IT(){
  spp(EXINT_PG);
  EIPR &= ~EIm_ipb0m;
  EIMR |= EIm_ib0m;
}

/*------------------------------------------------------------------
ROUTINE Name : INTI2C_EndTrans
Description :  Interrupt subroutine which occurs after an I2C end of
               transmission
Comments :   The "user part" presents the way to deal with read data from the
             I2C bus.
        (it reads 10 bytes from EEPROM and stocks them into EEPROM_REC[])
```

```
-----------------------------------------------------------------*/
#pragma interrupt( INTI2C_EndTrans )
void INTI2C_EndTrans(void){
  extern unsigned char EEPROM_REC[10];  /*already declared in main*/
  extern unsigned char ToRead;

  SAVE_PAGE;
/**** Beginning of User Part ****/

   ToRead--;       /* ToRead is the number of bytes which are left to read */
   spp(SPI_PG);
   EEPROM_REC[9-ToRead] = SPIDR; /* EEPROM_REC[] is the chart where read
                                     data are stocked */
   if(ToRead != 0){          /* If some data are left to read */
            I2C_ACK();       /* The ST9+ generates an ACK */
            I2C_READ_DATA();/* And then asks for a new read operation */
  }else{                     /* If all the data have been read */
            I2C_STOP();      /* The ST9+ generates a STOP */
            spp(EXINT_PG);
            EIMR &= ~EIm_ib0m; /* And then disables I2C interrupt*/
  }

/**** End of User Part ****/
   spp(EXINT_PG);
   EIPR &= ~EIm_ipb0m;     /* Clear flag */
   RESTORE_PAGE;
}
```

This is the Main program using the previous sub-routines:

```
/********************* EXTERNAL DECLARATIONS *********************/
#include ”i2c.h”
#include ”rccu.h”
/************************* MAIN PROGRAM*************************/
/* !!!!!!!!! INTCK should be 12MHz !!!!!!!!!!*/
```

```
unsigned char EEPROM_REC[10];
unsigned char ToRead;
/*---------------------------------------------------------------
ROUTINE Name :  main
Description :  This program :
            - writes chart display[] into an EEPROM connected with the ST9+
            using an I2C bus (I2C address : 0xA0)
            - then checks this write operation by reading the EEPROM.
            Read data are stored in chart EEPROM_REC[]
Comments : Read operations use end of transmission interrupts.
            Write operations use polling on the I2C bus to check if the bus
            is free before writing another byte.
---------------------------------------------------------------*/
void main(void)
{
   unsigned int counter;
   unsigned char display[10];
   unsigned char i;
   display[0] = 0xC0 ;  /* Fill source chart */
   display[1] = 0xF9 ;
   display[2] = 0xA4 ;
   display[3] = 0xB0 ;
   display[4] = 0x99 ;
   display[5] = 0x92 ;
   display[6] = 0x82 ;
   display[7] = 0xF8 ;
   display[8] = 0x80 ;
   display[9] = 0x90 ;
   INIT_PLL();        /* The PLL is configured for an INTCLK of 12MHz */
   INIT_I2C();
   INIT_I2C_IT();

                 /* Beginning of write operations */
   I2C_START();
   I2C_SEND_DATA(0xA0);  /*Slave address of EEPROM (Write operation)*/
   I2C_WAIT_ACK();
   I2C_SEND_DATA(0x00);  /*Internal address where data will be loaded*/
   I2C_WAIT_ACK();

   for( i=0 ; i<7 ;i++){ /* Between a START and a STOP, only 7 bytes can
                            be written in a row to the EEPROM */
           I2C_SEND_DATA(display[i]);
           I2C_WAIT_ACK();
   }
   I2C_STOP();
   I2C_START();
   I2C_SEND_DATA(0xA0);  /*Slave address of EEPROM  (Write operation)*/
   I2C_WAIT_ACK();
   I2C_SEND_DATA(0x07);  /*Internal address where data will be loaded*/
   I2C_WAIT_ACK();

   for( i=7 ; i<10 ;i++){          /*Last 3 bytes to send*/
           I2C_SEND_DATA(display[i]);
           I2C_WAIT_ACK();
   }
I2C_STOP();
/*We must wait 10ms per written byte before reading operations */
   for (counter = 0; counter < 0x5000; counter++)  I2C_WAIT();

                            /*Beginning of Reading operation*/
   I2C_START();
   I2C_SEND_DATA(0xA0);  /*Slave address of EEPROM (Write operation)*/
   I2C_WAIT_ACK();
   I2C_SEND_DATA(0x00);  /*Internal address where data will be read*/
   I2C_WAIT_ACK();

   I2C_START();             /*I2C Specification : it is a restart */

   I2C_SEND_DATA(0xA1);  /*Slave address of EEPROM (Read operation)*/
   I2C_WAIT_ACK();
```

```
    ToRead = 10;                /* 10 bytes are to read */
    Enable_I2C_IT();            /* Enable IT */
    I2C_READ_DATA();            /* Read the first byte */

    while(ToRead != 0);         /* During this loop, data are read using
                                interrupts */
     /*END OF PROGRAM*/
}
```

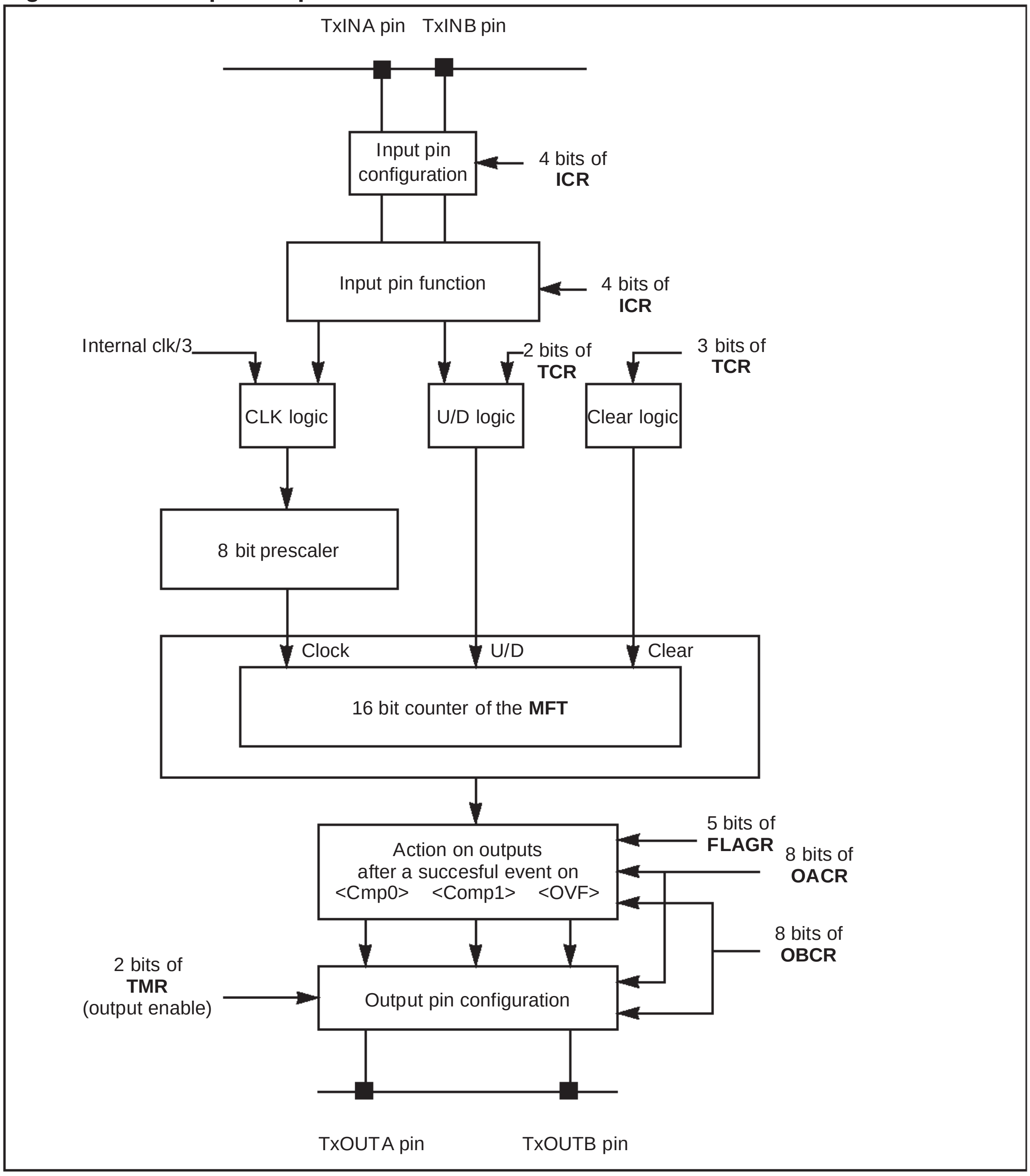
4.5 MULTIFUNCTION TIMER

This device is the most powerful of the ST9+ on-chip peripherals. This description only covers its main features. You can study the more intricate details with the help of the ST90158 - ST90135 Family Data Sheet.

The Multifunction Timer can handle many different operating modes; so many in fact, that virtually the only limit is your imagination. Six different applications are described here. Let's first have a look at the general organisation.

The first diagram in Figure 33 represents the inputs and outputs, the prescaler register and the clock selection blocks, with their associated configuration registers.

Figure 33. MFT Input/Output Modes

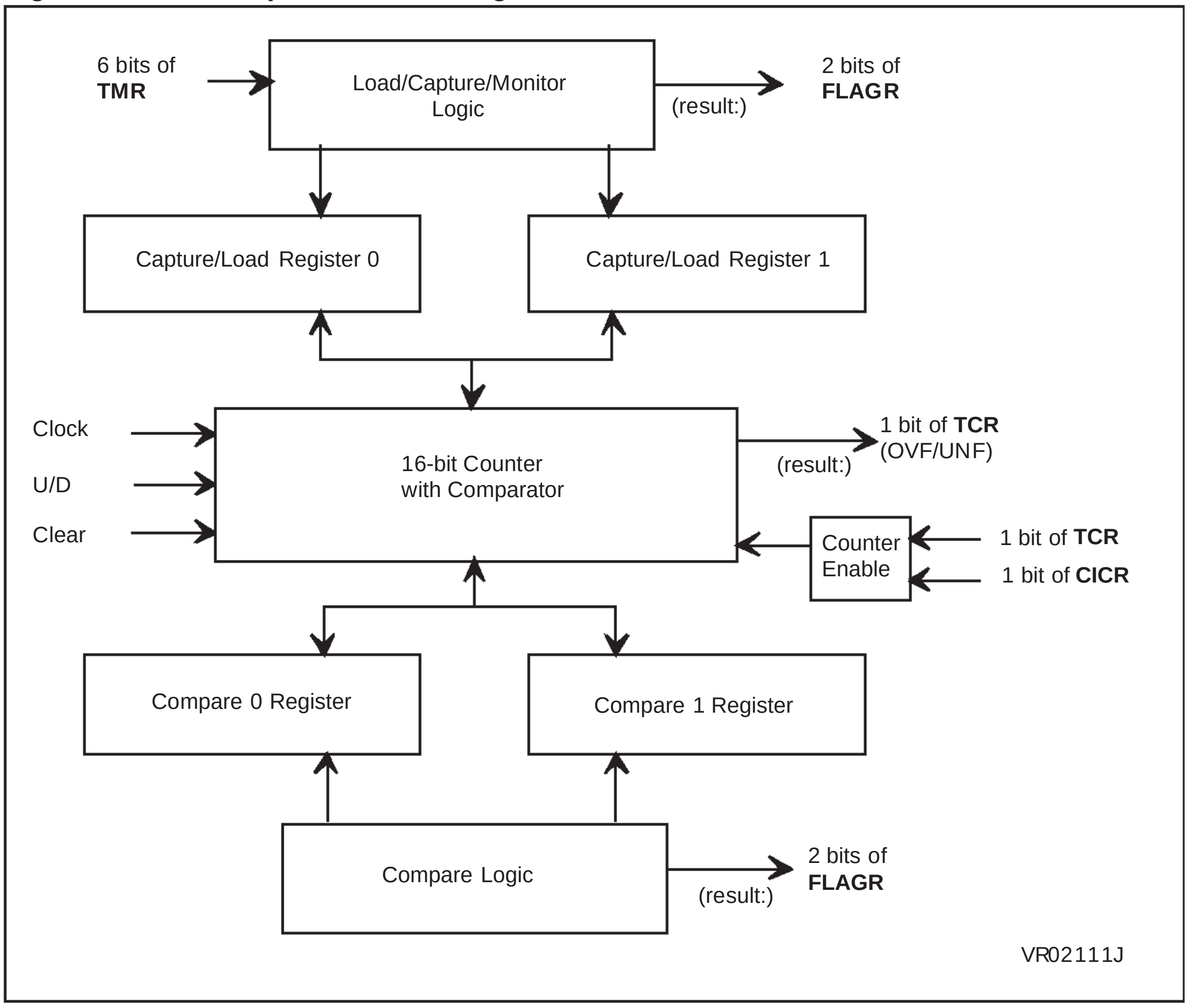


The two outputs are actually the Q outputs of a flip-flop which set and reset. Flip-flop inputs can be individually configured to receive pulses from either of the following events: compare with CM0, compare with CM1, and overflow/underflow. This allows generation of single-shot or periodic wave forms simply using the timer.

In addition to provide output signals, the timers can generate so-called ”internal events” that can be used to synchronise other internal peripherals such as the DMA, and the Analog to Digital Converter. Not any peripheral to be synchronised can be connected to any MFT. For each ST9+ variant, the connection between each MFT and the other peripheral is specified. Refer for this information to the corresponding Data Sheet.

The second block diagram represents the counter, the capture and compare registers, the reload logic and the associated configuration and status registers. The interrupt and DMA blocks are not represented.

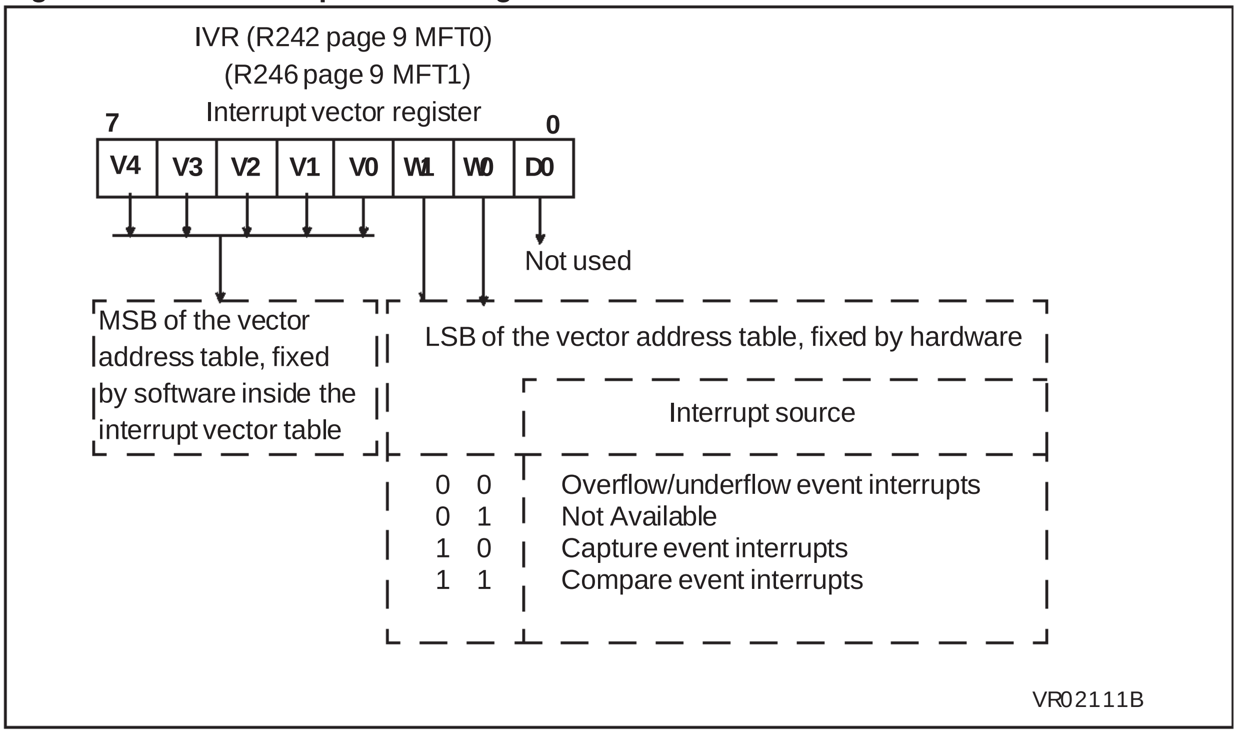
Figure 34. MFT Simplified Block Diagram



In the above diagrams, the interrupt and DMA logic are not represented. They obey the general interrupt and DMA rules described earlier, and are controlled by three registers.

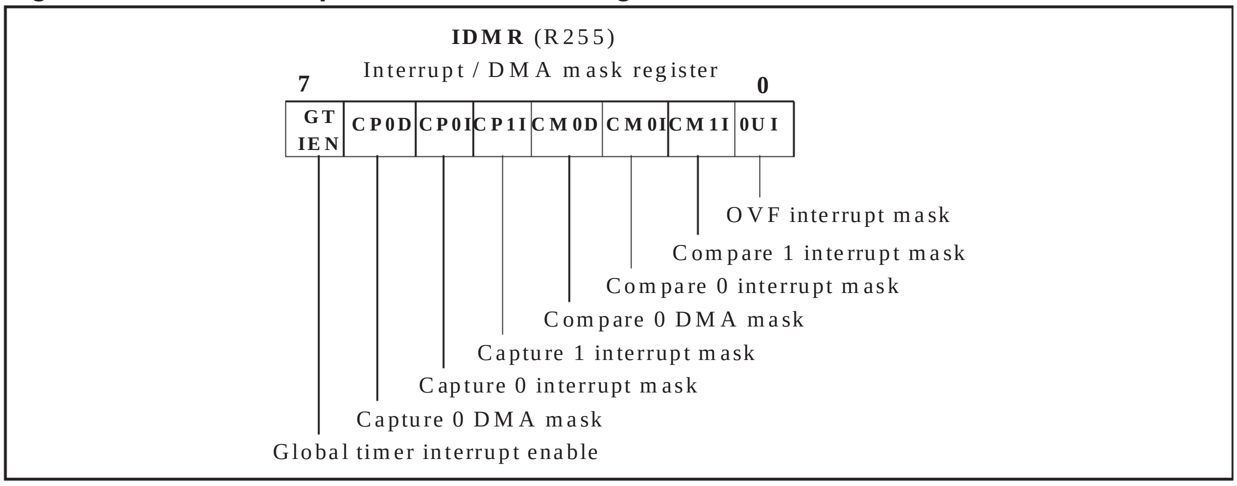
The Interrupt Vector Register controls the location of the interrupt vectors in program memory. They must be located at addresses that are multiples of 8.

Figure 35. MFT Interrupt Vector Register


IVR (R242 page 9 MFT0)
(R246 page 9 MFT1)
Interrupt vector register
7
0
V4
V3
V2
V1
V0
W1
W0
D0
Not used
MSB of the vector address table, fixed by software inside the interrupt vector table
LSB of the vector address table, fixed by hardware
Interrupt source
0 0 Overflow/underflow event interrupts
0 1 Not Available
1 0 Capture event interrupts
1 1 Compare event interrupts
VR02111B

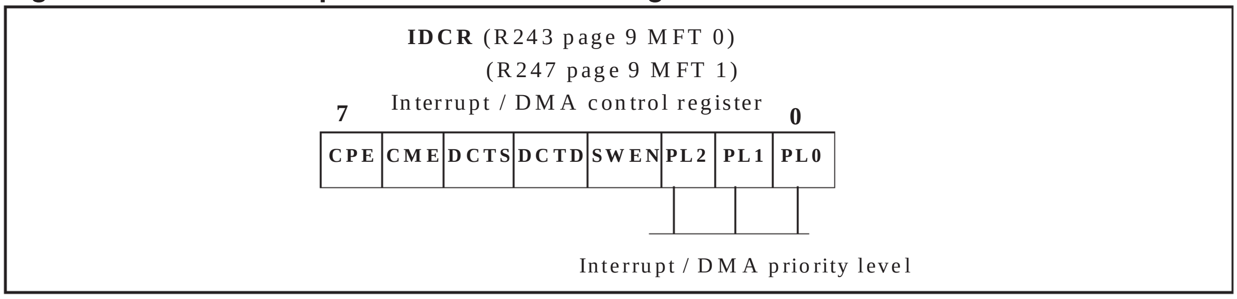
The Interrupt and DMA Mask Register individually enables and disables the various interrupt and DMA sources.

Figure 36. MFT Interrupt and DMA Mask Register



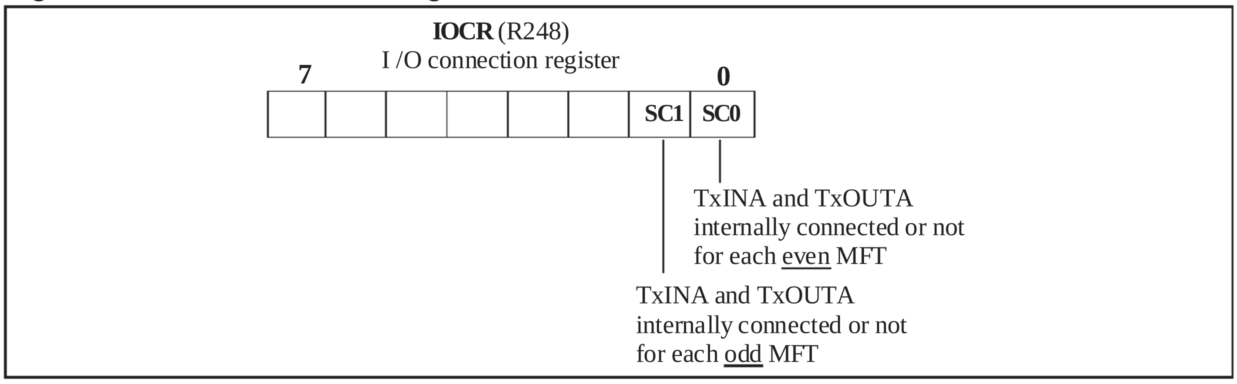
The upper five bits of the Interrupt and DMA Control Register indicate the status of the interrupts and the DMA blocks. The lower three bits set the Interrupt and DMA priority level.

Figure 37. MFT Interrupt and DMA Control Register



The Input and Output Control Register has only two active bits. They allow you to internally connect the Output A of each MFT to its own Input A. One bit does this connection for all even-numbered MFTs, and the other for all odd-numbered MFTs.

Figure 38. MFT I/O Control Register

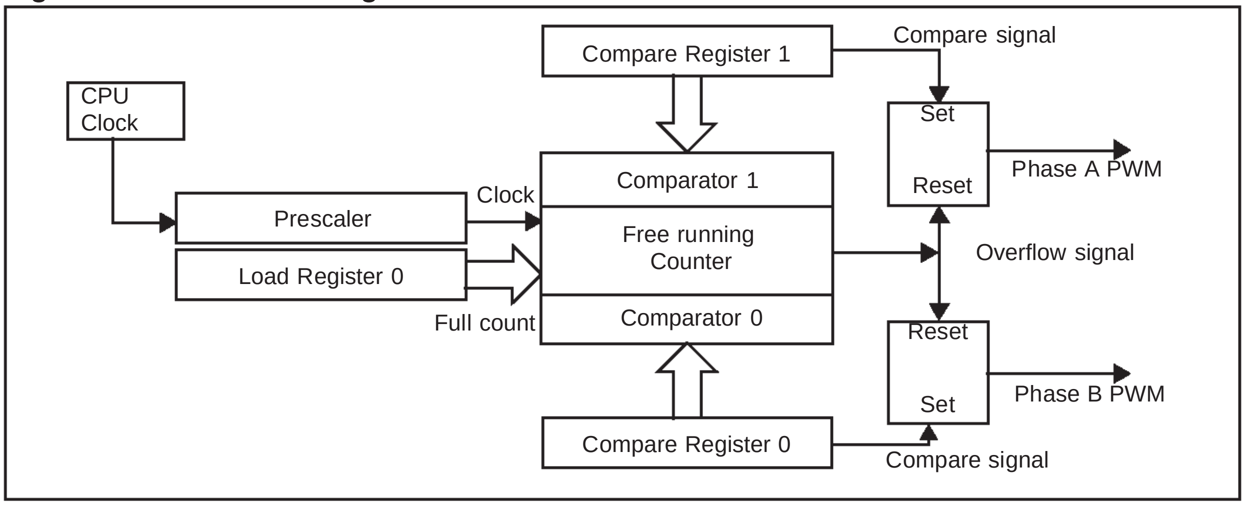

IOCR (R248)
I /O connection register
7
0
SC1
SC0
TxINA and TxOUTA internally connected or not for each even MFT
TxINA and TxOUTA internally connected or not for each odd MFT

4.5.1 Generating Two Pulse Width Modulated Waves with One MFT

4.5.1.1 Description example

The Multifunction Timer is used here as a double pulse-width modulator. It is also possible to have a single PWM output if needed. Let’s look at a simplified block diagram of the MFT showing only the functional blocks that are actually used:

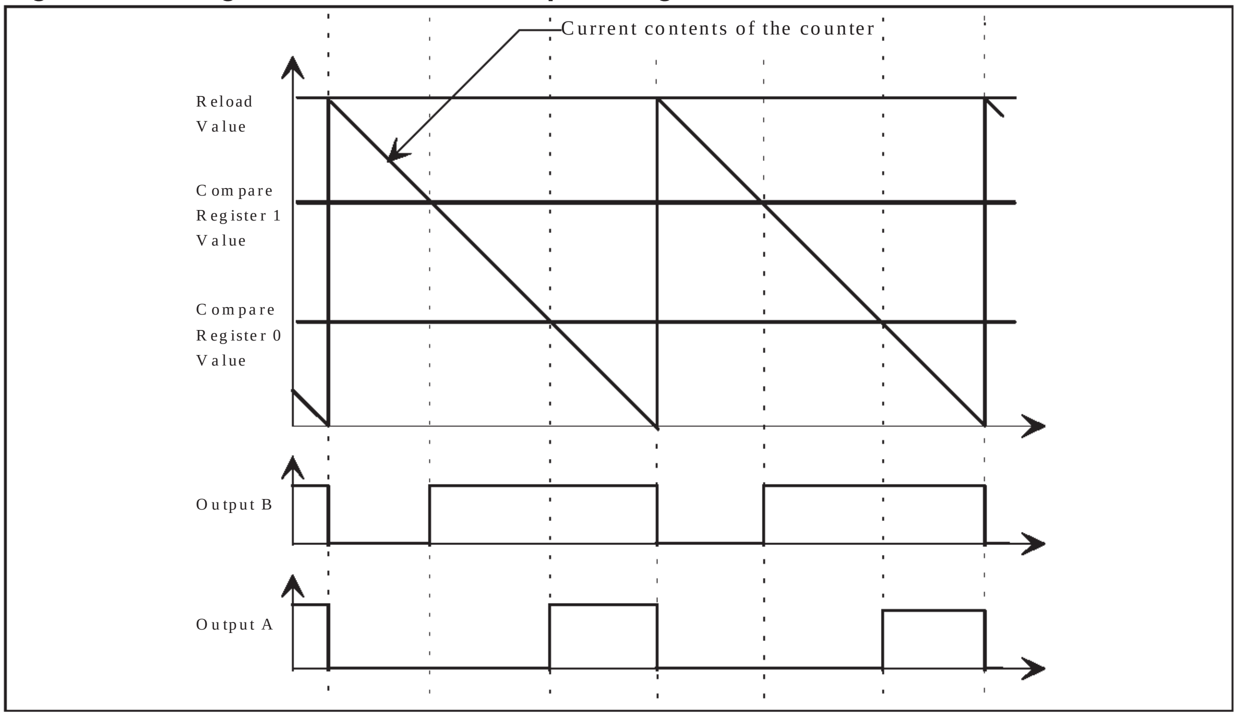
Figure 39. MFT Block Diagram



This is possible using the two comparators that simultaneously compare each capture register with the free-running counter. When the counter overflows, it is reloaded with the contents of the Load Register 0. At that time, both outputs are reset.

When the value of the counter becomes equal to one of the compare registers, a pulse is sent to the output flip-flop of the corresponding side, thus setting the output. So the low time is the time between the counter overflow and the comparison. The high time is the remainder of the period.

Figure 40. Using a Counter and 2 Compare Registers to Modulate 2 Pulse Widths



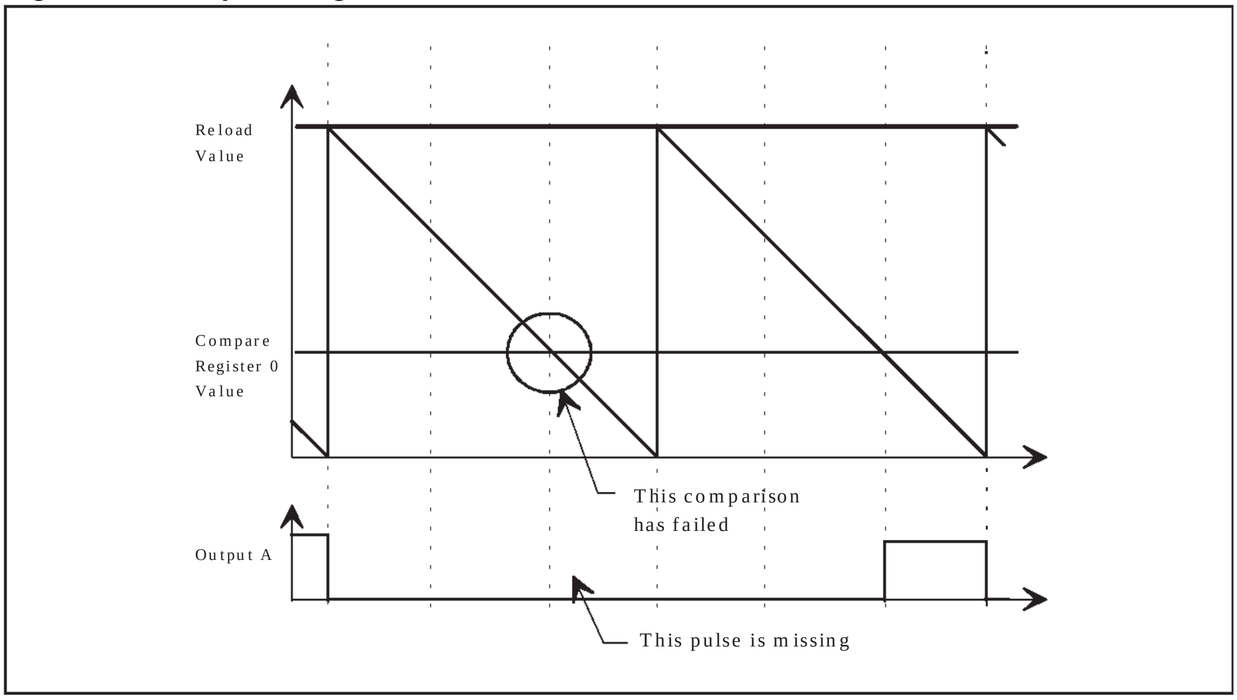
Actually, as nearly everything is configurable by values in control registers, this is only one of the ways to do it. In particular, the direction of the counter (upward or downward) and the effect of the actions on the outputs (set/reset) can be chosen at will.

This value of the Load Register determines the frequency of the output signals. For example, with a reload value of 255 and a 25 Mhz internal clock, the PWM frequency is 1/256 of the 25 MHz clock divide by 3, that is 32552 kHz. In this case, the range of the compare registers is 1 to 255 inclusive. A value of 0 locks the outputs to the high state.

To use the timer to deliver variable PWM, we need to change the value of the compare registers from time to time. The simple way that first comes to mind is to write into the compare register whenever we need to update the PWM rate. This can lead to problems.

If we write the new value into the compare register while the counter value matches the previous contents, this can make the comparison fail and the pulse that changes the state of the output is not produced. The result is the output signal misses one cycle, as follows:

Figure 41. Compare Register is modified while its Content matches Counter



This will obviously have a bad effect on the electronic circuit connected to this output.

Another problem can occur if the reload value exceeds 255. Then, two bytes are needed to express the compare values. Since the ST9+ is an 8-bit machine, the two bytes that make up the word are sent one after the other. If the high bytes of the old and new values are the same, there is no problem. But if they differ, there are three possibilities for the value in the compare register:

– The register contains the old value

– The register contains one byte of the old value, and the other byte of the new value

– The register contains the new value

Depending on the relative values of the old and new values, on the order of writing the bytes (high byte first, or low byte first), and on the direction of the timer (up-counting or down-counting), various things may happen. These can range from a missed comparison (see Figure 41) to two comparisons in the same cycle (not a problem).

This indicates that you have to pay attention to the time at which the compare register has to be updated. If we consider that the duty cycle does not vary widely from one cycle to another, the soonest after the match is the safest to change the value. The best way to do it is to store the new value in a variable (a register is a good choice), and configure the MFT to trigger an interrupt when the match occurs. Then, the interrupt service routine is executed and the new value is safely copied into the compare register.

The next section will give you an example of the MFT initialisation routine followed by the interrupt service routine that updates the compare register synchronously with the match.

4.5.1.2 Initialisation of the MFT for a twin PWM generator

```
/* *********** configure timer 0 for double pwm output ****************/
extern void * INTRELOADVECT (void) ; /* address of the pointer array to the
                                          intr. routine */
void ConfigTimer0 ( void )
   {
   SelectPage (T0D_PG) ;
   T_REG0R = 255 ;/* reload value. Provides a 256 clock pulse period PWM
   wave */
   T_CMP0R = 128 ;/* default duty cycle, side A*/
   T_CMP1R = 128 ;/* default duty cycle, side B*/
   T_TCR = (Tm_cen) ;/* start counter, down-counting */
   /* The counter will not start until Global Counter Enable is set in
   register CICR */
   T_TMR = (Tm_oe1+Tm_oe0) ;/* enable A and B outputs */
   T_ICR = 0 ;/* no input used */
   T_PRSR = 0 ;/* prescaler rate = 1 */
   T_OACR = (Tm_c0_set+Tm_c1_nop+Tm_ou_res) ;/* output A=1 on compare 0,
   0 on overflow */
   T_OBCR = (Tm_c0_nop+Tm_c1_set+Tm_ou_res) ;/* output B=1 on compare 1,
   0 on overflow */
   /* For the outputs to work, port 9 must be set to alternate function on
   P9.1 and P9.3 */
   T_FLAGR = 0 ;/* not used */
   T_IDMR = Tm_gtien + Tm_cm0i + Tm_cm1i ;/* interrupt enabled on compare,
   both sides */

   SelectPage (T0C_PG) ;/* MFTimer 0 control registers page */
   T0_IVR = (unsigned char)INTRELOADVECT ;/* Array of pointers to service
   routines */
   T0_IDCR = 0 ;/* highest priority for its compare interrupt */
   }
```

The timer is initialised and ready to start when the Global Counter Enable is set in register CICR. It refers to the array of interrupt vectors to the interrupt service routine shown in the next paragraph.

4.5.1.3 Interrupt Service Routine

Here is the declaration of the vectors in lower program memory. This text is inserted in the start-up module. It provides the INTRELOADVECT identifier as a global value for use in the initialisation routine above.

```
   .org 30h
INTRELOADVECT::
   .blkw   3                ; skip first 3 vectors
   .extern RELOADINT
   .word   RELOADINT        ; vector to timer 0 compare register interrupt
```

The code that follows is the interrupt service routine. There is only one vector for both compare events, so the routine tests which side has triggered the interrupt:

```
; INTERRUPT ROUTINE TO RELOAD THE COMPARE REGISTERS SAFELY
; *****************************************************
RELOADINT:
   .global RELOADINT
   push    r0               ; save one working register
   push    PPR              ; Save page pointer register
   spp      #T0D_PG         ; switch to page of data registers of timer 0
   ld      r0, T_FLAGR      ; flags of timer 0
   btjf    r0.5, NOCOMP0
   and      T_FLAGR, #~Tm_cm0        ; compare 0 occurred : clear flag
   ld       PwmA, R61                ; copy new value into compare register

NOCOMP0:
   btjf     r0.4, NOCOMP1

   and      T_FLAGR, #~Tm_cm1        ; compare 1 occurred : clear flag
   ld       PwmB, R62                ; copy new value into compare register

NOCOMP1:
   pop      PPR
   pop      r0
   iret                              ; end of interrupt
```

Note: The above routine takes into account that two matches could possibly occur at the same time, or even while the interrupt routine is in progress. In any case, if the interrupt returns while the other channel has matched, the same interrupt will be triggered again as soon as the return is complete.

4.5.2 Generating a Pulse Width Modulated Wave with a Cleaner Spectrum

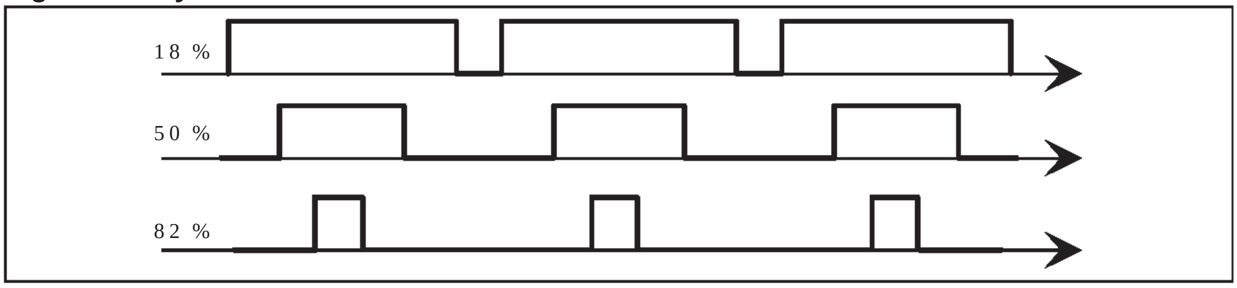
The use of a MFT to produce two PWM signals is convenient, especially for example in a stepper motor application. There, an ST90158 is used and only one MFT is available for PWM generation, since the other one is used for other purposes. However, this method suffers from a parasitic phase modulation of the signal since the falling edge is fixed, and the rising edge moves with the desired duty cycle. This leads to the production of unwanted harmonics in the resulting spectrum. This is not a problem for a lot of applications.

Sometime these harmonics are prohibited. Example: when we want to drive a high power induction motor, these harmonics have two serious drawbacks:

– They produce parasitic frequencies that are injected in the mains, which is not allowed by the power distribution companies,
– They feed the induction motor with even ranked harmonics, which degrade the efficiency of the motor and reduce the peak output power.

You can improve this situation by using a symmetrical PWM generation, producing waveforms as follows:

Figure 42. Symmetrical PWM Waveforms

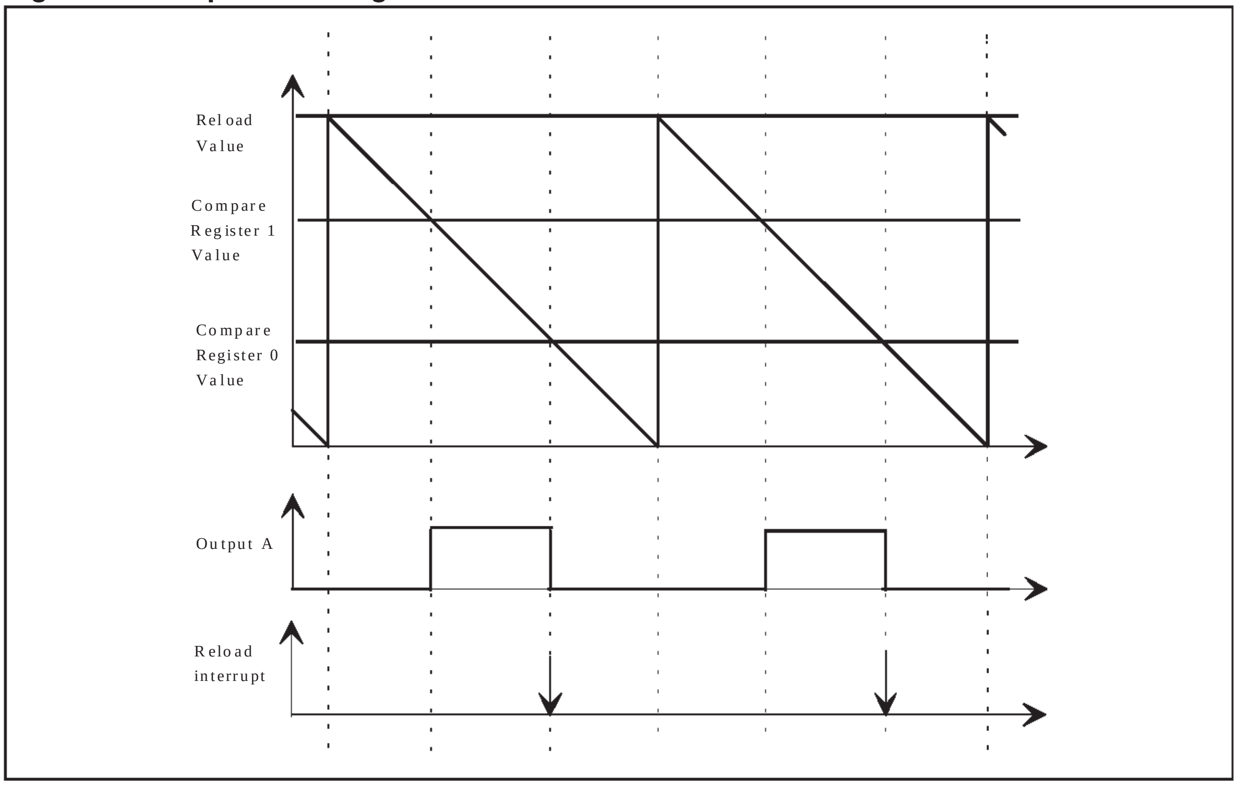


You use the MFT in the following way. One MFT being used for one output signal, you use only output A. You configure it so that it is set on a comparison with compare register 1, and reset on a comparison with compare register 0. These registers are always loaded with values that are symmetrical centred on half of the reload value. For example, if the reload value is 255, the compare registers are set to the following values:

Duty cycle	Compare register 0	Compare register 1
10%	115	140
25%	96	159
50%	64	191
75%	32	223
90%	13	242

As explained in Section 4.5.1.1, you need to ensure that the compare registers are not written while they are used for a comparison. Because of this, they are written in an interrupt service routine that is triggered by the MFT itself. To allow maximum latency time, the interrupt is triggered either by a compare 0 or a compare 1 event, according to the value of the duty cycle. For duty cycles between 0 and 50%, the compare 0 event is used. Between 50% and 100%, the compare 1 event is used. The timing diagram is the following:

Figure 43. Output A Timing



4.5.2.1 Initialising the MFT for Symmetrical PWM Signals

This initialisation routine does not include the initialisation of the corresponding port. You must set the bit corresponding to output A of the timer to alternate function. Also, since an interrupt is triggered for reload, you must define the interrupt vector table in ROM and the IVR register of the MFT appropriately. You should set the interrupt priority to a high level since the maximum latency allowed is a half-period of the timer.

```
; This routine configures one timer. Only the registers in TxD_PG page are
; set.
; The page number is set by the caller.
ConfigOneTimer:
   ldw     T_REG0R, #255          ; reload value. Provides a 255 clock
                                  ; pulse period PWM wave
   ldw     T_CMP1R, #64           ; default duty cycle, rising edge
   ldw     T_CMP0R, #192          ; default duty cycle, falling edge
   ld      T_TCR, #Tm_cen         ; start counter, down-counting
   ; The counter will not start until the Global Counter Enable is set in
   ;register CICR
```

```
ld      T_TMR, #Tm_oe0          ; enable A output
ld      T_ICR, #0               ; no input used
ld      T_PRSR, #3              ; prescaler rate = 4 (period = 256 us)
ld      T_OACR, #(Tm_c0_set+Tm_c1_res+Tm_ou_nop)
                                ; output A=0 on compare 0, 1 on compare 1
ld      T_OBCR, #(Tm_c0_nop+Tm_c1_nop+Tm_ou_nop)
                                ; output B not used
                            ; For the outputs to work, the appropriate
                            ; port must be set to alternate function
ld      T_FLAGR, #0             ; not used
ld      T_IDMR, #Tm_gtien + Tm_cm0i ; interrupt enabled on compare 0
                                    ; (will vary)
ret
```

4.5.2.2 Routine for Changing the Duty Cycle

Since the values of CMP0 and CMP1 must remain symmetrical relative to the value 127.5, each of these should change by one unit each time the control value changes by two units. This would lead to a loss in resolution, for only a total of 127 different values would be available. To remedy this, the two compare registers are not exactly symmetrical. When the control value changes by one unit, one of them changes by one unit, and the other remains at the previous value. Thus, the difference between the two values keeps the resolution of 256 different values, as shown in the table:

Control value	CMP0	CMP1	CMP1 - CMP0
-2	65	191	126
-1	64	191	127
0	64	192	128
1	63	192	129
2	63	193	130

The routine also selects the event (compare 0 or compare 1) that causes the next interrupt according to the sign of the control value.

```
; This interrupt routine reloads the counters when the comparison has
; occurred. This is necessary to prevent glitches in the output signals.
  The
; control value is stored in a register giving the voltage to apply on the
; load. This value is copied to the MFT Compare Register right after the
;comparison has occurred.
; Note: the control value is signed and ranges between -127 and +127.
```

```
ReloadPWM:
   .global ReloadPWM
   .interrupt
   pushw    rr0              ; save one pair of working registers
   push     PPR              ; Save page pointer register
   spp      #T0D_PG          ; switch to page of data registers
   ld       r1, Value        ; Voltage input value, -127 to +127
   sra      r1               ; Take half that value
   ld       r0, r1           ; Take a copy
   adc      r0, #0           ; Retrieve bit that fell out. r0+r1 = PhaseP
   add      r1, #192         ; Compute upper compare value
   ld       T_CMP0LR, r1     ; set falling edge

   ld       r1, #64
   sub      r1, r0
   ld        T_CMP1LR, r1    ; set rising edge

   ; check for sign of input value to attach interrupt request either to
   ; CMP0R or CMP1R comparison. If previous subtraction produces no carry,
   ; interrupt will be attached to compare 1.
   jrc      RLP1

   ld       T_IDMR, #Tm_gtien + Tm_cm1i ; interrupt enabled on compare 1
   pop      PPR
   popw     rr0
   iret

RLP1:
   ld       T_IDMR, #Tm_gtien + Tm_cm0i ; interrupt enabled on compare 0
   pop      PPR
   popw     rr0
   iret
```

4.5.3 Incremental Encoder Counter

An incremental encoder is a device that generates two square signals in quadrature when its shaft rotates. Its main specification is the number of cycles per revolution, i.e. the number of square signal cycles for one shaft revolution. The greater the number of cycles, the more accurate it is, and the more it costs.

In a stepper motor application, it is used to monitor the rotation of the motor. Here however, we have taken an encoder that has a great resolution, and this section will show you how to use an MFT to count encoder pulses, and rescale it to another resolution to meet your project needs.

4.5.3.1 Description

The MFT includes an input block, a counter block, an event block and an output block. In this application, no output signal is required, since the counter is expected to produce a value that is read in registers.

The functions the MFT must provide are:

Use the inputs to drive the counter according to the amount and direction of rotation

Keep the count undisturbed by events other than the transitions at the inputs

Allow reading the current counter value

Allow setting the counter to an arbitrary initial value

We configure the input block to use the two square signals delivered by the encoder to drive the up and down counting of the counter. We do this using a special mode of the input block called ”autodiscrimination”. In this mode, the input pulses clock the counter, and the phase relationship between the two signals selects up or down counting. Since the process is incremental, once the MFT configured this way, it automatically follows the rotation of the encoder shaft and the counter value reflects its position relative to the position it was in when the counter was configured. In our example, the encoder produces 512 cycles per revolution. Because the counter is 16-bit, it can monitor shaft positions 65536/512 = 128 (or +/-64) full revolutions from the original position.

To operate with an encoder as the input pulse source, we need to set the MFT to Continuous Mode. When the counter crosses the FFFF to 0000 boundary in either direction, this is called an End-Of-Count event. In the normal Continuous mode, the End-Of-Count event reloads the counter with the contents of REG0R (or REG1R in biload mode). In this application, no automatic reload may occur at any time so we can use the full range of the counter and be able to cross the boundary. To prevent the reload occurring, we must set the REG0R register to Capture Mode. The MFT is then said to be in Free-Running Mode.

We have thus selected the Free-Running Mode, yet we need to be able to set the counter to any arbitrary value that is taken as the initial position. For this we need to use the REG0R register as load register. The solution is to initialise the MFT with REG0R in capture mode by setting the RM0bit of the TCR register, to ensure Free-Running Mode. When we need to load a value in the counter, we temporarily clear bit RM0. Then we set REG0R to the value that must be written in the counter. Bit CP0 of register FLAGR is pulsed high to write the contents of REG0R into the counter. Then we set bit RM0 again.

4.5.3.2 Initialisation

The code for initialising the MFT is given below.

Note: The ICR register sets the input to autodiscrimination mode. In this mode, the lower four bits are irrelevant.

```
void ConfigTimer1 ( void )
   {
   SelectPage (T1D_PG) ;
   T_REG0R = 0 ;/* This register is used to preload the counter */
   T_REG1R = 0 ;/* This register is used to read the counter */
   /* T_CMP0R and T_CMP1R are not used */
   T_TCR = Tm_cen ;/* enable counter */
   /* The counter will not start until the Global Counter Enable is set in
   register CICR */
   T_TMR = Tm_rm0 ;/* capture using REG0R and monitor using REG1R */
   /* This mode is not desired, but is implies free-running, which we need
   */
   T_ICR = Tm_ab_aa ;/* inputs used in autodiscriminating mode */
   /* For so, the port 3 must be set to input on pins P3.4 and P3.6 */
   T_PRSR = 0 ;/* prescaler rate = 1 */
   T_OACR = 0 ;/* ouput A not used */
   T_OBCR = 0 ;/* ouput B not used */
   T_FLAGR = 0 ;/* not used */
   T_IDMR = 0 ;/* not used */
   }
```

4.5.3.3 Reading

To read the value of the counter, you need to capture it in one of the registers, since the counter itself does not have an address and cannot be read directly. There are two ways of reading the counter:

Set one of the registers REG0R or REG1R to capture mode. A pulse on bit CP0 or CP1, respectively, transfers the contents of the counter to the corresponding register.

Put REG1R in Monitor Mode. In this mode, REG1R continuously reflects the value of the counter, without the need for pulsing a bit in a register.

In the stepper motor application, we have chosen to use Monitor Mode. No special function is needed to read the counter. A simple assignment of REG1R to a variable is enough, such as:

```
   Pos = REG1R ;/* Get current counter value */
```

4.5.3.4 Updating

As mentioned in Section 4.5.3.1, updating requires switching the mode of REG0R from Capture to Load.

```
/* This function forces the MFT1 counter to a value given in argument */
void SetEncoder ( int Value )
   {
   SelectPage (T1D_PG) ;
   T_TMR &= ~Tm_rm0 ;/* Switch temporarily to mode ”Load with REG0R” */
   T_REG0R = Value ;/* set encoder */
   /* convert motor position to match the encoder’s resolution */
   T_FLAGR |= Tm_cp0 ;/* Load counter with this value by pulsing CP0 high */
   T_FLAGR &= ~Tm_cp0 ;/* Return CP0 to low */
   T_TMR |= Tm_rm0 ;/* Go back to mode ”Capture with REG0R” that implies
   free-running */
   }
```

4.5.4 Light Pen Capture

4.5.4.1 Application Description

The Multifunction Timer is very useful for various measurement tasks. In this application, it will be used to read a barcode label.

A barcode label is a very popular means of labelling products for automatic reading, with or without the intervention of an operator. It is made of a series of black lines printed on a clear background. The width of the black lines and white spaces varies and this codes the information. Many different coding systems are available today, but all have in common that the width of the bars and of the spaces only have two values: narrow and wide, meaning binary coding. By moving a scanning device across the label, the succession of lines and spaces can be converted into a succession of zeroes and ones, that just need to be interpreted using a decoding algorithm.

This algorithm is described in the barcode reader application in Section 4.5.5. All we cover in this section is the way to capture the bar widths.

In this application, we shall use an optical wand that will be handled manually. Any other scanning system, like a flying-spot, could be used as well.

The analog output of the wand is converted to binary through a simple threshold circuit. The output of this circuit is fed to input A of the MFT.

4.5.4.2 MFT Configuration

The MFT is configured as follows:

The clock source is taken from the internal clock, and the prescaler is set so that the full count is reached in about half a second.

Input A is made sensitive to both rising and falling edges. Each of these triggers a capture of the current value of the counter, and resets the counter to zero.

We set the DMA system so that each value captured is written to memory. This is done without the intervention of the processor core.

As long as transitions are received that are less than captured. At the end of the scan, there is necessarily a time-out condition. The MFT is set to trigger an interrupt on this condition. This interrupt request attention from the core. It can then start processing the list of time values that have automatically been stored.

If the wand is idle, it will supply no transition at all and the core will be interrupted every half a second. It is then easy to check that the list of times is empty, meaning that no label has been read.

Thus, the only overhead on the processor is being interrupted twice a second to do a very simple check if no data is available. At the same time, the current level of the reading device is determined (black or white).

4.5.4.3 Initialisation Routine

The following assembler source code defines a table of 100 words which is sufficient to accommodate the capture of the whole label. It also defines the registers that are used for the DMA transaction counter and pointer. Another register is a flag that will be set to one by the time-out interrupt routine if some data has been captured.

The initialisation routine sets the Input Control Register (ICR) so that input A is used as a trigger, is sensitive to both rising and falling edges, and input B is unused. The Interrupt/DMA Mask Register enables the Global Timer Interrupt Flag, enables the Overflow/Underflow Flag as an interrupt source, and the CaPture 0 as a DMA trigger. The Timer Control Register is set for Up Counting, and Clear on Capture. However, the Counter Enable bit is not yet set. Thus, a transition on input A will capture the counter value, transfer it to memory using DMA, and clear the counter.

When the counter overflows, it triggers an interrupt.

The Prescaler Register is set for an input clock of 133.33 kHz allowing for about 0.5 seconds before overflow.

```
DMACounter = 100; RR100
DMAPointer = 104; RR104
DataFlag = 103; R103 is a 1 if data has been received.
DMABufferSize = 100

; Table of read values. After the program has run, it must contain the same
   values.
   .bss
   .global Captures
Captures:
   .blkw DMABufferSize ; Room for captures

; Initialisation of the MFT for DMA capture on both edges of inA
; and interrupt on overflow.
   .text
   .global InitMFTBarCode
InitMFTBarCode:
   spp #T1C_PG; MFT 1 control register page
   ld       T1_DCPR, #DMACounter   ; transaction counter. Buffer is in
                                   ; memory.
   ld       T1_DAPR, #DMAPointer + 1; DMA pointer ; target is data memory
   ld       T1_IVR, #OVFVect       ; Direct interrupts to this
                                   ; vector in ROM
   ldw      RR#DMACounter, #DMABufferSize*2 ; Reset DMA
   ld       R#DataFlag, #0         ; No data received yet
   ld       T1_IDCR, #4            ; DMA on capture enabled
                                   ; level = medium priority
   spp      #T1D_PG                ; MFT 1 data register page
   ld       T_TCR, #Tm_ccp0 + Tm_udc ; up counting, clear on capture
                                   ; started later
   ld                  T_TMR, #Tm_rm0 ; Capture into REG0R
                                   ; run continuously
   ld       T_ICR, #Tm_exa_rf + Tm_ab_ti ; Input A rising+falling
                                   ; sensitive, triggers capture
   ld       T_PRSR, #29            ; to clock main counter with 133.33 kHz
   ld        T_IDMR, #Tm_gtien + Tm_cp0d + Tm_oui ; Enable int ; int on over-
   flow ; DMA on capture
   ld       T_FLAGR, #0            ; Reset all errors
   ret
```

4.5.4.4 DMA Start Routine

This routine sets the DMA Pointer and Counter Registers to their initial values and sets the IDMR and the TCR, enabling the counter to work in DMA mode. Note that the DMA Counter Register is set to twice the number of captures, since a capture involves two bytes (high and low bytes of the 16-bit timer register).

```
; This routine starts the DMA transfer by initialising the DMA registers.
   .global StartDMA
StartDMA:
   ldw      RR#DMAPointer, #Captures ; Point to beginning of buffer
   ldw       RR#DMACounter, #DMABufferSize*2 ; Enable DMA
   spp      #T1C_PG                ; MFT 1 control register page
   ld       T1_IDCR, #4            ; Clear possible End of Block condition
   spp      #T1D_PG                ; Select MFT 1 data register page
   ld       T_FLAGR, #0            ; Reset all errors
   ld      T_IDMR, #Tm_gtien + Tm_cp0d + Tm_oui ; Enable int ; int on over-
   flow ; DMA on capture
   ld      T_TCR, #Tm_cen + Tm_ccp0 + Tm_udc  ; up counting, clear on
                                              ;capture
                                              ; start now
   ret
```

4.5.4.5 Time-Out Interrupt Routine

This routine is called each time the counter overflows. When no capture has occurred, the DMA transaction counter remains at its initial value. In this case, the interrupt routine merely clears the interrupt request and returns. When one capture at least has occurred, the routine disables the DMA and clears the interrupt request. It sets the DataFlag to signal that some data is available.

It is up to the main program to process the records. When is done, it resets the DataFlag and calls the StartDMA routine to re-enable further captures.

```
; This interrupt service routine checks whether some data is available
; if yes it sets the flag. It then restarts DMA.
   .global OVFInterrupt
OVFInterrupt:
   push    PPR              ; It is modified in this routine
   cpw     RR#DMACounter, #DMABufferSize*2 ; At least one transfer ?
   jpeq    OVF1

   spp     #T1D_PG          ; Select MFT 1 data register page
   and     T_TCR, #~Tm_cen  ; Stop timer until data processed
   spp     #T1C_PG          ; Yes. select MFT 1 control register page
   ld      T1_IDCR, #Tm_cpe+4 ; Set End of Block condition to stop DMA
   ld      R#DataFlag, #1   ; Set data received flag

OVF1: spp #T1D_PG           ; Select MFT 1 data register page
   and     T_FLAGR, #~(Tm_ouf+Tm_ocp0+Tm_ocm0) ; Clear interrupt request
   pop     PPR
   iret                     ; and also possible overruns
```

4.5.5 Combined Use of a Single MFT

The Multifunction Timer, as we have seen is so rich in possibilities that only a very fertile imagination could exhaust all of them. This means that you can use a single MFT for several functions at the same time, when at first glance, several timers seem required.

This paragraph gives you a hint how this could be done. It has not been implemented in real life, but such an application is certainly feasible. This can suggest other combinations to you that can save a timer and maybe allow you to use a smaller MCU type in the ST9+ family than the one you first considered.

4.5.5.1 Application Description

This application regulates the speed of a motor that spins a mirror for a flying-spot barcode reader. In many department stores, the barcode label of the product is read remotely by a laser beam that scans a portion of space near the cash register. The cashier has just to bring the item within that area to give it a chance to be caught by the flying spot. A photodiode or a photomultiplier receives the reflected light. The signal is processed to recognise a possible barcode modulation.

The flying spot is produced by deflecting the beam of a fixed laser using a mirror arrangement that spins at constant speed and that is designed to scan the space in much the same way as the picture is scanned in a television camera.

In this application, a D.C. motor spins the mirror. We have to regulate the motor’s speed. A speed-regulation loop is not sufficient, since there is always an error in the speed.

If we want to ensure a speed that has an absolute value with no error, we must use a phase-locked loop. Then, there can be an error on the phase (which is the angular position of the shaft) but the mean speed is exactly the set speed.

To drive the motor, we shall provide a pulse-width modulated signal. This has been described previously (See Section 4.5.1). The timer is reloaded after a total count of N, giving a reload frequency of

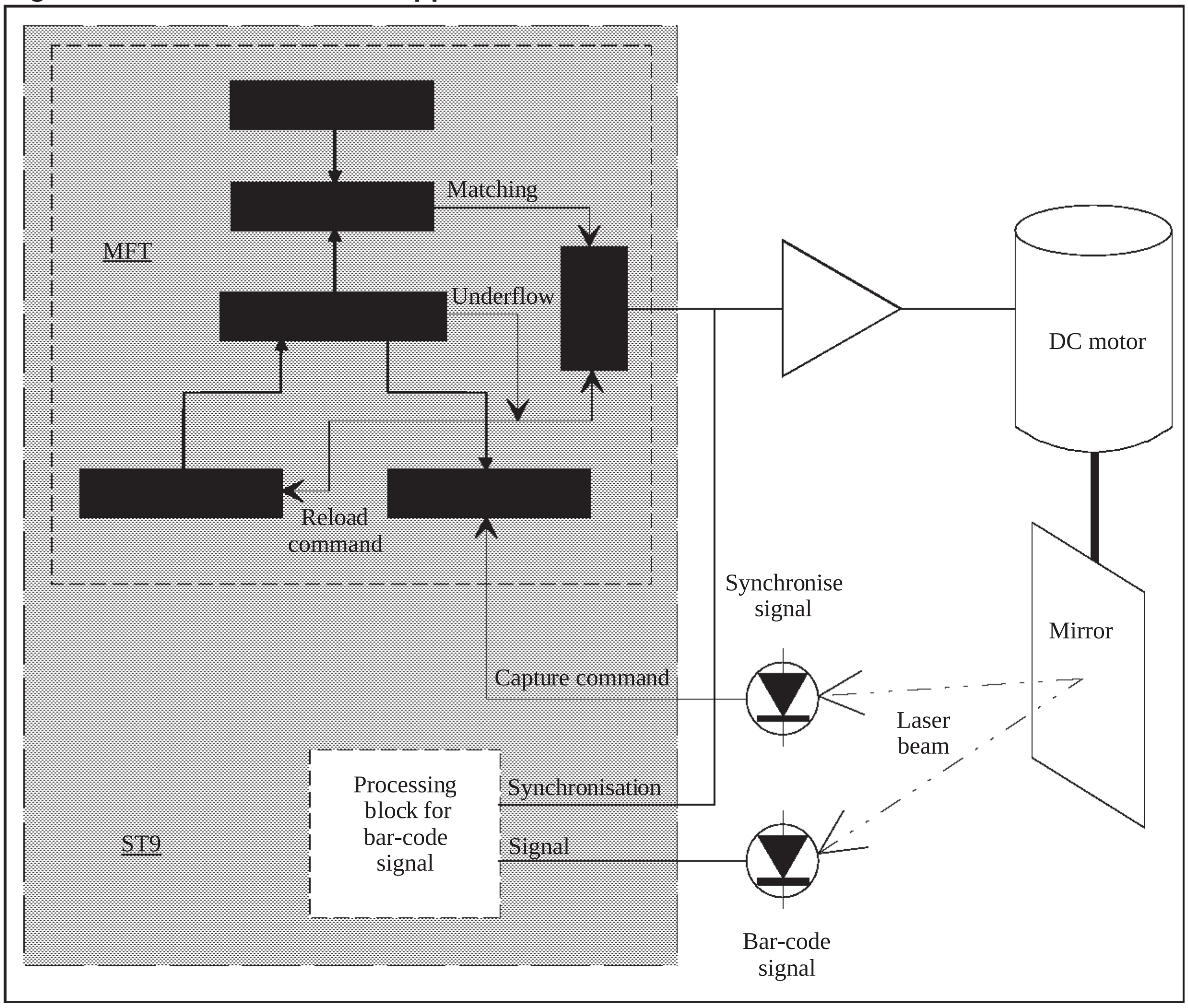
$$\int reload = \frac{\int clock}{N}$$

The flying-spot scans the area using a succession of horizontal lines, these lines being arranged vertically to fill the whole area. We want to set the frequency of the horizontal scan to exactly the frequency of the reload of the counter, as given above. To measure the phase, the optical system has a photodiode installed so that it is hit by the laser beam once at the end of one horizontal scan line.

The signal of the photodiode is used to trigger a capture of the timer count. Since the frequency of this signal is exactly that of the timer reload, the number read expresses the phase of the light pulse relative to the timer reload. It is then possible to regulate this phase by closing the loop.

The counter reload frequency being that of the horizontal scan, the reload produces a change of an output pin that is externally connected to the input of another timer that will capture of the video signal that contains the bar code information.

Figure 44. Barcode Scanner Application



4.5.5.2 Initialisation Code

The following code initialises the timer so that it will provide PWM on its output A and use input A to trigger a capture. Both a comparison with register 0 and a capture into register 1 will produce an interrupt request.

```
/* ******** configure timer 0 for pwm output and capture ***************/
extern void * INTRELOADVECT (void) ; /* address of the pointer array to the
intr. routine */
void ConfigTimer0 ( void )
   {
   SelectPage (T0D_PG) ;
   T_REG0R = 255 ;/* reload value. Provides a 255 clock pulse period PWM
   wave */
   T_CMP0R = 128 ;/* default duty cycle */
   T_TCR = (Tm_cen) ;/* start counter, down-counting */
   /* The counter will not start until Global Counter Enable is set in
   register CICR */
   T_TMR = Tm_oe0 + Tm_rm1 ;/* enable A output, load from REGOR, capture
   into REG1R */
   T_ICR = Tm_ab_ti + Tm_exa_f ;/* A trigger, B not used, A falling-edge
   sensitive */
   T_PRSR = 0 ;/* prescaler rate = 1 */
   T_OACR = (Tm_c0_set+Tm_c1_nop+Tm_ou_res) ;/* output A=1 on compare 0,
   0 on overflow */
   /* For the output to work, port 4 must be set to alternate function on
   P4.7 for ST90158*/
   T_FLAGR = 0 ;/* not used */
   T_IDMR = Tm_gtien + Tm_cm0i + Tm_cp1i ;/* interrupt enabled on compare 0
   and capture 1 */

   SelectPage (T0C_PG) ;
   T0_IVR = (unsigned char)INTRELOADVECT ;/* Array of pointers to service
   routines */
   T0_IDCR = 0 ;/* highest priority for its compare interrupt */
   }
```

The timer is initialised, but will only start when the Global Timer Enable of the CICR register is set.

4.5.5.3 Interrupt Service Routine

Here, the vectors are positioned in the lower program memory. This text is inserted in the start-up module. It provides the INTRELOADVECT identifier as a global value for use in the above initialisation routine.

```
   .org    30h
INTRELOADVECT::
   .blkw   2           ; skip first 2 vectors
   .extern CAPTUREINT
   .word   CAPTUREINT ; vector to timer 0 capture interrupt
   .extern RELOADINT
   .word   RELOADINT  ; vector to timer 0 compare interrupt
```

Next comes the interrupt service routine. There is only one vector for both compare events, so the routine tests which side has triggered the interrupt:

```
; INTERRUPT ROUTINE TO RELOAD THE COMPARE REGISTERS SAFELY
; *******************************************************
RELOADINT:
   .global RELOADINT
   push    r0               ; save one working register
   push    PPR              ; Save page pointer register
   spp     #T0D_PG          ; switch to page of data registers of timer 0
   ld      r0, T_FLAGR      ; flags of timer 0
   btjf    r0.5, NOCOMP0

   and     T_FLAGR, #~Tm_cm0; compare 0 occured : clear flag
   ld      PwmA, R61        ; copy new value into compare register

NOCOMP0:
   btjf    r0.4, NOCOMP1

   and     T_FLAGR, #~Tm_cm1; compare 1 occurred : clear flag
   ld      PwmB, R62        ; copy new value into compare register

NOCOMP1:
   pop     PPR
   pop     r0
   iret                     ; end of interrupt
```

4.6 SERIAL COMMUNICATIONS INTERFACE

4.6.1 Description

The SCI is the association of a UART and a complex logic that handles tasks such as character recognition and DMA. It can also work as a simple serial expansion port that then resembles the SPI. The example given here appears as a classical UART application, viewed externally, but it is assisted by the built-in DMA controller to provide effortless transfers of data in and out of the application. As serial communication is a feature that is very often used, it will be found in all three applications. For example, refer to the files \programm\smcb\config.c and \programm\smcb\serial.c.

The SCI has four operating modes:

– Asynchronous mode where data and clock can be asynchronous. Each data is sampled 16 times per clock period. The baud rate should be set to the 16 division mode and the frequency of the input clock is set to suit.

– Asynchronous mode with synchronous clock where data and clock are synchronous but the transmit and receive clock are asynchronous. The receive clock must be given by the external peripheral.

– Serial expansion mode where data and clock are synchronous and the clock is supplied by the transmitter (ST9+ in transmission and externally in reception).

– Synchronous mode where data and clock are synchronous, the transmit data clock is supplied by the ST9+ and the receive data clock is received by the external peripheral. In this mode there are no start and stop bits.

These four operating modes allow you to connect the ST9+ to any external interface.

4.6.1.1 UART

The UART offers all the usual functions for asynchronous transfer. You can set it to handle word lengths of 5 to 8 bits, with or without parity, even or odd. It offers also the possibility to append a signalling bit (called Address bit or 9th bit--whatever the word length) at the end of the transmitted data, just before the stop bit. This can be used to help design a multidrop network. The UART can be set to interrupt the processor only when this bit is high, indicating that the current character is an address or other identifier. This prevents the processor being disturbed by the traffic on the network unless an address byte is received, so that the processor can check if it is concerned or not by the incoming data.

The UART is set to 8 bits, 1 stop bit and no parity, by writing the appropriate value in the CHCR register.

A timer is part of the SCI block. It is used as a BAUD Rate Generator. It allows you to divide either the internal clock or the frequency available at the RXCLK input pin, by an arbitrary value. This output can be fed to the receive and transmit sections, though each section can have its own clock frequency supplied on RXCLK and TXCLK for receive and transmit, respectively. Here, we use the BRG to clock both sections.

At the clock input of the transmit and receive shift registers, two other dividers can be inserted, to further divide the frequency by 16. It must be used in asynchronous mode to allow detection of the start bit using a local clock. If an external clock is supplied, at the same frequency and phase as the serial incoming data, the 16x divisor must not be used. Here, the predivisor is used.

The UART is surrounded by logic that allows it to detect errors on reception, a break state on the line, and also to recognise a match between the character received and a reference character stored in a register. This last feature is used here. The ACR register is loaded with the value 10 representing the LF character, and the IMR interrupt mask register is set to enable an interrupt on a character match. Since DMA is used to transfer the incoming characters to memory, this technique allows us to detect the end of message, if we state that all commands must end with an LF character.

4.6.1.2 DMA controller

The DMA allows automatic data transfer from memory or a register to the serial transmitter or from the serial receiver to a register or memory, or both.

The working of the DMA in conjunction with the UART merits some detailed explanation, because it is subtle and you need to take some precautions to make it work properly. Once initialised, it is easy to use and it consumes very little computing power to initialise transmission or re-enable reception.

The DMA transfers in each direction are controlled by two registers: the DMA transaction counter and the DMA pointer. The transaction counter is used to stop the transaction when a predefined number of characters are transferred. The DMA pointer points to the character to send or to the location that will hold the next character to be received. To initialise a DMA transfer, the transaction counter must be set to the number of characters to transfer and the DMA pointer must be set to the beginning of the storage area that contains the data to transmit or that will hold the data string received.

Transfers can occur between the UART and either memory or register file.

The transaction counter and the DMA pointer must be registers in the register file. If the register file is involved in the transfer, both the transaction counter and the pointer must be single bytes (the addressing range of the register file is 0 to 255). If memory is involved, the counter and the pointer must be each a pair of registers, since the addressing range in memory is 0 to 65535 for one segment.

Once you have defined in which registers you will put the counter and the pointer, you must let the SCI know. The SCI has two registers for this purpose called the DMA Address Pointer

Register (DAPR) and DMA Counter Pointer Register (DCPR). These registers are not the actual counter and the pointer, they point to where you want to locate them in the register file. This double pointer mechanism is similar to that described for interrupts.

Here there can be two cases: the transfer involves the memory or the register file.

If the transfer involves the memory, it's standard procedure. The DCPR points to the 16-bit Counter register, and the DAPR points to the 16-bit Address register. These two register pairs may reside anywhere there is room for them in the register file. Since register pairs are used, their addresses are even. The DCPR must be even. To reach the DMA segment the DMASR (or ISR depending on DAPR bit 0) MMU register contains the segment number.

If the transfer involves the register file, things are a bit different. Both the transaction counter and the address register are 8-bit values. They are supposed to be adjacent in the register file, i.e. they occupy two successive registers: the first one, even numbered, is the address register. The second one, odd numbered, is the transaction counter. To select this mode, the least significant bit of the DCPR must be one. Its value is the address of the address register (which is even) plus one. The DAPR is not used.

The whole mechanism described above applies for both the transmitter side and the receiver side. Thus, there are twice as many registers as mentioned: the TDCPR and the TDAPR for the transmitter side, the RDCPR and the RDAPR for the receiver side. The registers you have located in the register file are also twice as many. So, the TDCPR points to the transmitter transaction counter and the TDAPR points to the transmitter address register (with the exception mentioned above if the transaction involves the register file). The RDCPR points to the receiver transaction counter and the RDAPR points to the receiver address register.

When a DMA transfer is in progress, the processor is not even aware of it (except that it is slightly slowed down by the stolen clock cycles). So nothing special has to be done as far as the program is concerned. The question is: how to start and end a DMA transfer? Here, the processor is involved, and some code is needed. Depending on which side is concerned, the processes are different.

4.6.1.2.1 Transmitter Side

To start a DMA transmission, you need to do the following operations, in order:

– Set the DMA segment register DMASR (or ISR) to point on the wanted segment of data memory.

– Set the transmit address register to the address of the second word to send (not the TDAPR, which is only set once at initialisation, and points permanently to the transmit address register).

– Set the transmit transaction counter to the number of characters to transmit minus one (not the TDCPR, which is also only set once at initialisation, and points permanently to the transmit transaction counter).

– Set the TXD bit of the IDPR register. This bit enables the Transmitter holding register empty flag (TXHEM bit of register ISR) to trigger a DMA transfer when the last character is sent. When TXD is reset, the holding register empty flag would trigger an interrupt instead.

– Load the Transmit Buffer Register (TXBR) with the first character to send. This starts the DMA process as soon as this character is transmitted.

– Clear the Transmitter holding register empty flag (TXHEM bit of register ISR) and the Transmitter buffer register empty flag (TXSEM bit of register ISR) to prevent DMA transfer starting before the first character is fully transmitted.

– Enable the end of DMA interrupt by setting the Transmitter Data Interrupt Mask (TXDI bit of register IMR).

The DMA is now started and will stop when the transaction counter reaches zero. The code that does these operations is given in Section 4.6.3.

Note: The transmit address register is initialized with the address of the **second** word and the transmit transaction counter is initialized to the number of characters to transmit **minus one,** because DMA transfer begins when the first word is transmitted (a write in the Transmit Holding Register), so the first word is transmitted ’by hand’.

4.6.1.2.2 Receiver Side

The receiver side is a little more complicated as two kinds of events can occur during reception:

– There can be transmission errors that must be handled.

– A break is received or a character match event can occur.

The receiver is set in a ready for DMA condition by the initialisation code. See the example of an initialisation routine below.

Reception normally finishes when the DMA transaction counter has reached zero. But this is not necessarily the way one wants to use the serial input, since it implies that the number of characters to receive must be known before the transfer starts. In most cases, character-type messages have a variable length.

In this application, the transaction counter is not expected to reach zero, since the size of the buffer assigned for reception exceeds the longest message defined in the specification. Thus, if the transaction counter reaches zero this is in fact an overflow condition. In this case, the contents of the buffer are discarded and a new reception is initiated. In both cases, the steps to perform an end of block are:

– Reset the Receiver End of Block (RXEOB bit of register IMR) to remove the interrupt request.

– Make use of the received characters if so desired.

– Re-enable the end of block interrupts by setting the Receive DMA Bit (RXD bit of register IDPR).

– Restore the DMA capability by setting the receive address pointer (not the RDAPR) to the address of the beginning of the receive buffer.

– Set the receive transaction counter (not the RDCPR) to the number of characters to receive. This will re-enable the DMA capability.

In the an example application, commands messages could have variable length and be terminated by an LF character. So, we use the character match event capability built in to the SCI.

If the Address/Data Compare Register (ACR) contains the value 10 (ASCII code for LF), and the Receive Address Mask (RXA bit of register IMR) is set, whenever an LF character is received, an interrupt is generated. This interrupt is considered in the application as a good message received signal. It is processed the following way:

– In this application, no other characters are expected until the previous message has been answered by the program. However, it may be useful to inhibit further DMA cycles by clearing the RXD bit in the register IDPR and also the RXDI bit in the register IMR.

– Read the Receive Buffer Register (RXBR) to remove the interrupt request (see note below)

– Make use of the received characters.

– Restore the DMA capability by setting the receive address pointer (not the RDAPR) to the address of the beginning of the receive buffer.

– Set the receive transaction counter (not the RDCPR) to the number of characters to receive. This will re-enable the DMA capability.

Note: When matching occurs between the incoming character and the ACR register contents, the DMA request is not generated, neither is an interrupt request for ”character received”. So, the incoming character remains in the receive buffer and no other character can be received until it has been removed. This is why it is not sufficient to clear the RXAP bit in the ISR register to reinstate the DMA transfer. You must clear the RXBR register by reading it during the character match interrupt service routine.

Finally, an error may occur during transmission. This can be a parity error (if parity checking is enabled), a framing error or an overrun error. If one of these errors occurs, take the following steps:

– Clear the bits indicating an error condition in the ISR register.

– Restore the DMA capability by setting the receive address pointer (not the RDAPR) to the address of the beginning of the receive buffer.

– Set the receive transaction counter (not the RDCPR) to the number of characters to receive. This will re-enable the DMA capability.

4.6.1.2.3 Interrupt Vectors

The interrupt vector scheme has been explained in Section 3.4.1 using the example of the MFT to illustrate the mechanism. Here is the equivalent table for the SCI giving the position of the four vectors dedicated to the four interrupt causes.

Value in IVR	40

Interrupt cause	**Address in ROM of the selected pointer**	**Value of the Pointer to interrupt routine**
Receiver error	40	Address of error processing routine
Break detect or address match	40+2	Address of Address match processing routine
Receiver data ready or DMA end of block	40+4	Address of receive DMA end of block processing routine
Transmitter buffer empty of DMA end of block	40+6	Address of transmit DMA end of block processing routine

4.6.2 Initialisation Code Example

```
/** configure Serial Communication Interface for DMA on both directions **/
void ConfigSCI ( void )
   {
   SelectPage( MMU_PG ) ;
   DMASR = 0x20 ;/* RR249 = DMA segment register point on data memory*/
   SelectPage( SCI1_PG ) ;
   S_RDCPR = 96 ;/* RR96 = receive counter */
   S_RDAPR = 98 ;/* RR98 = data pointer */
   S_TDCPR = 100 ;/* RR100 = receive counter */
   S_TDAPR = 102 ;/* RR102 = data pointer*/
   S_IVR = (unsigned char)INTSCIVECT ;/* Interrupt vector */
   S_ACR = LF ;/* contains the <LF> character to compare with incoming
   characters */
   S_IMR = Sm_rxa + Sm_rxdi + Sm_rxe ;/* Interrupts: <LF> found+end trans-
   mit block+error */
   S_IDPR = 0x14 ;/* medium priority for the sci interrupt, receive DMA */
   S_CHCR = Sm_wl8 + Sm_sb10 + Sm_am ;/* 8 bits, 1 stop, no parity, enable
   charact. match */
   S_CCR = 0 ;/* internal 16x clock */

   /* with a 12 MHz internal clock, a divisor of 78 */
   /* gives an asynchronous bit rate of 9600 with an error of less than 0.2
   % */
   S_BRGR = 78 ;
   /* The serial in an out pins must be configured on port 7 */
   /* Initialisation of the DMA control registers. */
   ReceiveDMACounter = RECEIVE_BUFFER_LENGTH ;
   ReceiveDMAPointer = ReceiveBuffer ;
   TransmitDMACounter = 0 ;
   }
```

4.6.3 Start Transmission Function

```
/* Start Transmission. */
void StartTransmission ( int Count )
   {
   TransmitDMAPointer = &TransmitBuffer[1] ;
   TransmitDMACounter = Count-1 ;/* Start transmission */
   /* -1 because the first character is send directly */
   SelectPage( SCI1_PG ) ;
   S_IDPR |= Sm_txd ;/* Set bit TXD to start DMA */
   S_TXBR = TransmitBuffer[0] ;
   S_ISR &= ~(Sm_txsem | Sm_txhem ) ;/* Reset transmitter empty bits */
   S_IMR |= Sm_txdi ;/* Enable end of DMA block interrupt */
   }
```

4.6.4 Interrupt Service Routine for End Of Block on Reception

If you want to transfer fixed-length character strings, you can use the same routine and add code to make use of the data received before re-initialising the DMA function.

```
#pragma interrupt( ReceiveOverFlow, 8)
void ReceiveOverFlow ( void )
   {
   EI() ;   /* enable high-level interrupts */
   PushCurrentPage() ;
   SelectPage( SCI1_PG ) ;
   S_ISR = 0 ;/* Reset all errors and do nothing */
   S_IMR &= ~Sm_rxeob ;/* Reset end of block bit */
   S_IDPR |= Sm_rxd ;/* Restore DMA capacity */
   ReceiveDMAPointer = ReceiveBuffer ;
   ReceiveDMACounter = RECEIVE_BUFFER_LENGTH ;
   PopCurrentPage() ;
   }
```

4.6.5 Interrupt Service Routine for Character Match on Reception

```
#pragma interrupt( CharacterMatch, 8)
void CharacterMatch ( void )
   {
   EI() ;   /* enable high-level interrupts */
   PushCurrentPage() ;
   SelectPage( SCI1_PG ) ;
   S_ISR = 0 ;/* Reset all */
   * ReceiveDMAPointer = S_RXBR ;/* **Important** */
                                            /* Remove the character that pro-
   duced the interrupt */
   * ReceiveDMAPointer = ’\0’ ;/* Terminate string */
   MessageReceived = TRUE ;/* Flag incoming message */
   ReceiveDMAPointer = ReceiveBuffer ;
   PopCurrentPage() ;
   }
```

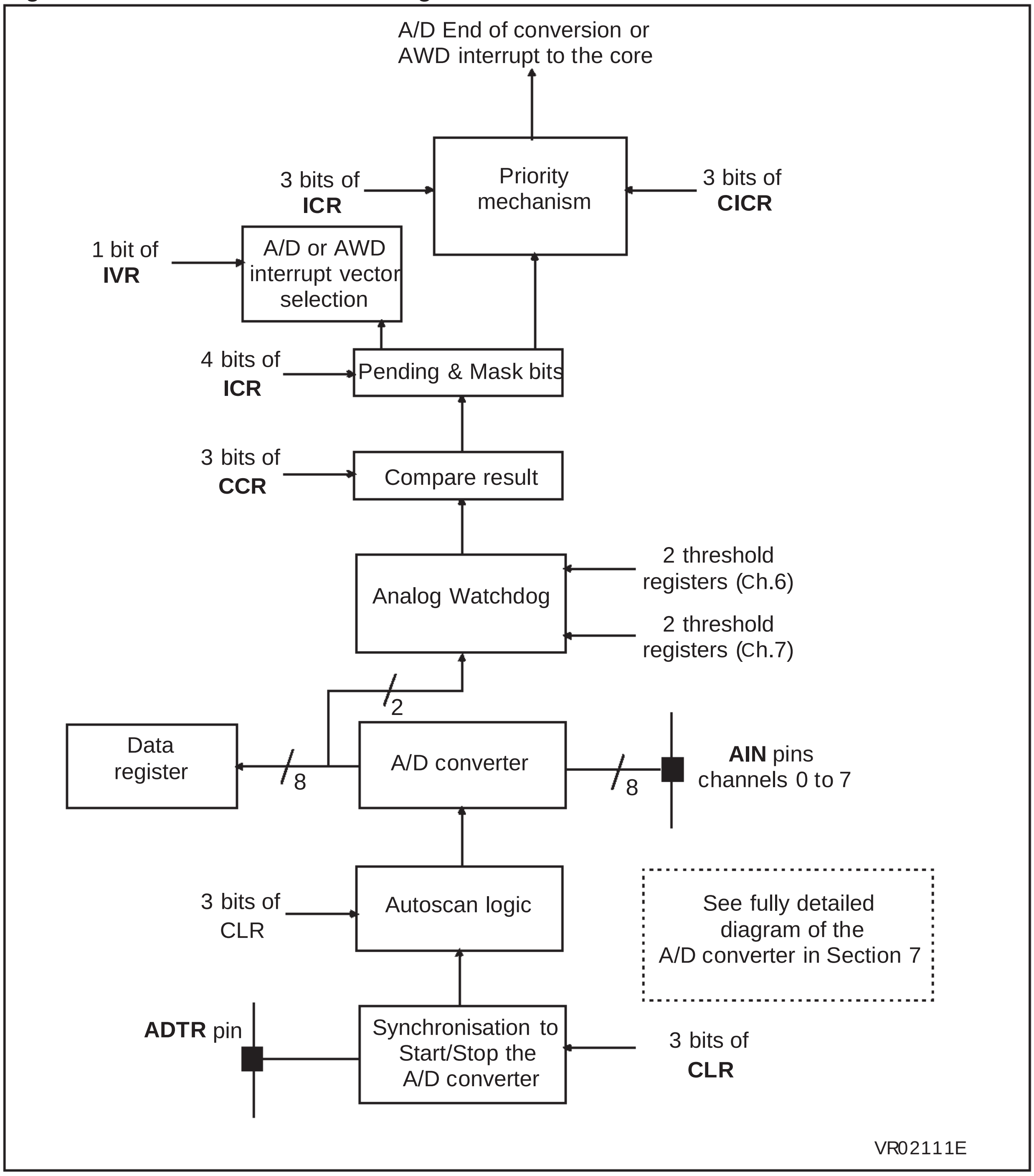
4.7 ANALOG TO DIGITAL CONVERTER

4.7.1 Description

The analog to digital converter is one of the simplest peripherals of the ST9+ family to use. It converts the voltage applied to one of eight inputs using an 8-bit successive approximation analog to digital converter. According to the value of the bits SC2, SC1 and SC0 of the Control Logic Register (CLR), one to eight inputs are converted at each conversion cycle, starting at the channel number specified by these bits and ending with channel 7.

The simplified block diagram is the following:

Figure 45. A/D Converter Block Diagram


A/D End of conversion or AWD interrupt to the core
Priority mechanism
3 bits of ICR
3 bits of CICR
1 bit of IVR
A/D or AWD interrupt vector selection
4 bits of ICR
Pending & Mask bits
3 bits of CCR
Compare result
Analog Watchdog
2 threshold registers (Ch.6)
2 threshold registers (Ch.7)
2
Data register
8
A/D converter
8
AIN pins channels 0 to 7
3 bits of CLR
Autoscan logic
See fully detailed diagram of the A/D converter in Section 7
ADTR pin
Synchronisation to Start/Stop the A/D converter
3 bits of CLR
VR02111E

The conversion cycle can either be started in Single Mode, by a trigger (software or hardware, the latter being internal or external) or in Continuous Mode, restarted as soon as the previous cycle is finished.

Each channel uses one bit of a port (the port depends on the variant of the ST9+), and the conversion result is stored in a register. Eight such registers are available, named D0R to D7R.

The end of conversion sets bit ECV of the ICR register, and can trigger an interrupt request, if the ECI bit (mask of the end of conversion interrupt request) of the Interrupt Control Register (ICR) is set.

Notes The internal interrupt controller does not automatically reset the ECV bit. It must be reset by the interrupt service routine prior to returning from interrupt. Failure to do this would cause the interrupt to loop endlessly.

As mentioned in Section 4.2 on parallel ports, you must configure the corresponding pins to alternate function both to reduce loading on the analog source and to avoid excessive dissipation in the pin’s input buffer. For the same reason, it is advised to keep this I/O configuration if an analog voltage is present on the pin, even if it is not being converted at this time.

4.7.2 Analog Watchdog

A special feature of this peripheral is the so-called Analog Watchdog. If enabled, the values of D7R or both D6R and D7R are each compared after each conversion with a pair of bounds stored in registers LT6R and UT6R, for D6R, and LT7R and UT7R, for D7R. If the value is outside the bounds, the AWD bit of the ICR register is set. This can trigger an interrupt request if the AWDI bit of the same register is set. To know which of the four bounds has been crossed, the Compare Result Register (CRR) has four bits each corresponding to one of these bounds. The appropriate bit(s) is/are set when the crossing has occurred.

Notes: The AWD bit, like the ECV bit, is not automatically reset by the internal interrupt controller. Thus it must be reset by the interrupt service routine prior to returning from interrupt. Failure to do this would cause the interrupt to loop endlessly.

To permit power saving in applications where energy conservation is important, you can power the ADC on and off so that it only consumes power when used. However when you switch it on, allow a delay of 60 µs can occur. The reset condition is off.

4.7.3 Interrupt Vectoring

Two vectors are dedicated to the ADC: the end of conversion vector and the analog watchdog vector. This allows these two events to be serviced by two different routines. Which of these vectors is used depends on the state of the W1 bit of the IVR register. This bit is set or reset according to the source of the interrupt.

However, it may occur that both interrupts are generated simultaneously. In that case, the analog watchdog takes over.

4.8 PERIPHERAL INITIALIZATION

All the ST9+ peripherals have a large number of configuration options. This makes them highly adaptable but requires a lot of complex initialization.

To help you to use the peripherals easily, you can find all the initialization programs on the <http://www.st.com> internet site.

The purpose of these programs is to give you C language programs to get started with programming each peripheral. This program package is composed of: a Header file, a peripheral function, a startup and a main program for each peripheral.

The startup (crt9.asm) and the main (main.c) are not explained here. Have a look at these files for more details. All you now have to know, is that whenever an interrupt subroutine is used, the startup program is initialized with the interrupt subroutine address.

For example the INTADC_EndConv() ADC interrupt subroutine address declared in the file <adc.c> is loaded in the startup routine <crt9.asm> at the Interrupt Vector Register place.

4.8.1 The initialization header file

The initialization header file defines all the constants, the file to include, the function prototypes and the peripheral mode.

The initialization header file has to be included any time the peripheral is referenced.

The next section uses the Analog to Digital Converter peripheral file as example.

The ADC initialization header file is <adc.h>.

4.8.1.1 Constants already initialized and to be initialized

The <adc.h> file is complementary to the header file <ad_c.h> located in the GNU9P\INCLUDE.ST9 directory like the other peripheral header files.

The constant initialization is done to initialize the non initialized constants used in the application.

E.g.: To use only one channel (channel 7).

```
/*-*-*-*-*-*-*-*-* CONSTANTS ALREADY INITIALIZED -*-*-*-*-*-*-*-*-*-*/
#define AD_MASK 0x00
#define TOO_LOW 0x00
#define TOO_HIGH 0xFF

#define CHANNEL0 0x00
```

```
#define CHANNEL1 0x20
#define CHANNEL2 0x40
#define CHANNEL3 0x60
#define CHANNEL4 0x80
#define CHANNEL5 0xA0
#define CHANNEL6 0xC0
#define CHANNEL7 0xE0
/*-*-*-*-*-*-*-*-* CONSTANTS TO BE INITIALIZED -*-*-*-*-*-*-*-*-*-*/
#define AD_CHANNEL CHANNEL7/*The lowest number of the AD channels used
                     For example, if you use channels 4 and 7, enter:
                     #define AD_CHANNEL CHANNEL4
                     */
```

The 8 initialized constants CHANNEL0 to CHANNEL7 give you the 8 possibilities of initializing the non initialized constant AD_CHANNEL. This constant will initialize the ADC Control Logic Register AD_CLR in the <adc.c> file.

4.8.1.2 The files to include

The files to include contain other initialized constants like in the previous file <ad_c.h>.

4.8.1.3 Function prototype declarations

Each of these functions can be called by the main program <mainc.c> or other functions for managing the ADC. They are defined in the <adc.c> file.

The <adc.h> file function prototypes are:

```
void INIT_ADC(void);
void START_ADC(void);
void STOP_ADC(void);
void INIT_ADC_IT(void);
void Enable_ADC_IT(void);
```

4.8.1.4 Defining the functional mode

To initialize the peripheral mode, you can activate the define directive by removing the <</* */ >>.

e.g.: In the file <adc.h> you can select continuous mode with:

```
#define continuous
/*#define single*/
```

4.8.2 The peripheral function file

These files contain the peripheral management functions. The function prototypes are placed in the header file as explained before.

The functional mode and the constants initialized in the header file modify the functions so that they can be used directly by the application program.

4.8.2.1 ADC example

As example the INIT_ADC function of the <adc.c> file is described:

```
/*-----------------------------------------------------------------
ROUTINE Name: INIT_ADC
Description:  Choice between Continuous and single shot modes.
       Possibility to initialize the analog watchdog on channel 6 or/and 7
Comments:    - A DELAY OF 60µs IS NECESSARY BEFORE USING THE ADC (delay
   included in this routine)
-----------------------------------------------------------------*/
void INIT_ADC(void)
{
  unsigned char x;

  spp(P7C_PG);        /* Initialization of port 7 as alternate function
                         (ST90158) */
  P7C2R = 0xFF;
  P7C1R = 0xFF;
  P7C0R = 0xFF;

 counter =0;           /* Specific initialization for this application */

  spp(AD0_PG);
  AD_CLR &= AD_MASK;
  AD_CLR |= ADm_pow;    /* Power up */

  for(x=0;x<10;x++);   /*  compulsory delay after power up */

  AD_CLR |= AD_CHANNEL;/* Number of simultaneous conversions :  */

  #ifdef continuous    /* continuous mode */
  AD_CLR |= ADm_cont;
  #endif

  #ifdef single           /* single shot mode */
  AD_CLR &= ~ADm_cont;
  #endif

  #ifdef awd6             /* analog watchdog on channel 6 */
  AD_LT6R = LOWER_BOUND;
  AD_UT6R = UPPER_BOUND;
  #endif

  #ifdef awd7             /* analog watchdog on channel 7 */
  AD_LT7R = LOWER_BOUND;
  AD_UT7R = UPPER_BOUND;
  #endif
}
```

The initialized constants AD0_PG, AD_MASK and ADm_pow have been initialized in the header file <ad_c.h>.

The initialized constants AD_CHANNEL, continuous, single, awd6 and awd7 have been initialized in the header file <adc.h>.

With the continuous mode and Analog Watchdog on channel 7 initialized, the <#ifdef> directive, after compilation, gives a program equivalent to:

```
void INIT_ADC(void)
{
  unsigned char x;

  spp(P7C_PG);          /* Initialization of port 7 as alternate function
                             (ST90158) */
  P7C2R = 0xFF;
  P7C1R = 0xFF;
  P7C0R = 0xFF;

  counter =0;            /* Specific initialization for this application */
  spp(AD0_PG);
  AD_CLR &= AD_MASK;
  AD_CLR |= ADm_pow;    /* Power up */

  for(x=0;x<10;x++);    /* compulsory delay after power up */

  AD_CLR |= AD_CHANNEL;/* Number of simultaneous conversions : */
  AD_CLR |= ADm_cont;   /* continuous mode */
  AD_LT7R = LOWER_BOUND;/* analog watchdog on channel 7 */
  AD_UT7R = UPPER_BOUND;
}
```

4.8.2.2 Peripheral function files presentation

4.8.2.2.1 ADC file

The ADC function file is <adc.c> and contains 6 functions initialized to use channel 7 of the ADC corresponding to I/O port P77 on ST90158.

It performs 15 conversions every 0.7s (nearly) and stocks the converted values in chart measures.

Moreover, analog watchdog is enabled on this channel and stops the series of conversions if the value is not within the prescribed thresholds.

ADC.C functions

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_ADC(void);	Choice between Continuous and single shot modes. Possibility of initializing the analog watchdog on channel 6 or/and 7.
void START_ADC(void);	Choice between three triggers to start the conversions: internal, external or software.
void STOP_ADC(void);	This routine stops the ADC after a series of conversions. After it, all power consuming logic is disabled (low power idle mode).
void INIT_ADC_IT(void);	The pointer to the array of two IT vectors dedicated to the AD is initialized. The priority of these interrupts is initialized. This routine initializes both end of count and analog watchdog. Don't forget to initialize your start-up file correctly.
void Enable_ADC_IT(void);	This routine enables interrupts: End of Conversion or/and Analog watchdog.
void INTADC_AnaWd(void);	**Interrupt subroutine** dedicated to Analog Watchdog.
void INTADC_EndConv(void);	**Interrupt subroutine** which occurs after an end of conversion event.

4.8.2.2.2 SPI file

The SPI function file is <i2c.c> and contains 10 functions initialized to perform data exchange between the ST90158 and an EEPROM connected on an I C bus at the I C address 0xA0/ 0xA1. Both read and write operations are performed:

– First, it writes the contents of chart display into the EEPROM.

– Then, it checks this write operation by reading the EEPROM using interrupts.

Read data are stored in chart EEPROM_REC.

The ports used for this application are:

– P7.1 SDI

– P9.7 SDO

– P9.6 SCL

I2C.C functions.

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_I2C (void);	Port initialization:(ST90158!) - P9.6: SCL (I2C Clock) - P9.7: SDO (I2C Data) These two pins are initialized as: ALTERNATE FUNCTION, OPEN DRAIN TTL The other pins of port9 are: BIDIRECTIONAL, WEAK PULL UP, TTL -P7.1: SDI (I2C Data) This pin is initialized as INPUT. The SPI is configured to work with I2C protocol For the moment, I2C bus remains disabled (clock not generated). If INTCLK > 12.8 MHz, you must change the value of constant I2C_SPEED in the i2c.h file.
void I2C_WAIT(void);	This is a loop of 4.7µs (including call and ret cycles) With an INTCLK of 12MHz, we need 57 cycles. This routine is written for INTCLK = 12MHz
void I2C_START(void);	This generates a start signal for I2C communications. While SCL remains high, SDA turns from high to low level.
void I2C_STOP(void);	This generates a stop signal for I2C communications. While SCL remains high, SDA turns from low to high level.
void I2C_WAIT_ACK(void);	This function waits for an ACK from the slave. To perform an ACK, the slave must force SDA to Low level. The ST9+ polls the SDA pin configured in input, during ”timeout”. If an ACK does not occur during this time, a STOP is generated, and the communication ends. If an ACK occurs before the end of timeout, the process leaves the function, ready to send new data.

FUNCTION	DESCRIPTION AND COMMENTS
void I2C_ACK(void);	This function generates a software ACK from the ST9+: During 4.7µs, SCL remains high while SDA is forced to low level. Used in read operations.
void I2C_SEND_DATA(unsigned char sendbyte);	Unsigned char sendbyte: the byte to send to the slave (data, I2C address...) sendbyte is sent on the I2C bus. This function can be used after: I2C_START() or I2C_WAIT_ACK(). The process remains in the function while the Busy Flag remains in the HIGH level (the transmission is not finished). If your software uses interrupts to send data on I2C bus, delete the last line of this function (polling).
void I2C_READ_DATA(void);	0xFF is written in the data register to start the transmission. This function must be followed by I2C_ACK(). If your software doesn't use interrupts to read data from I2C bus: -Check that the busy flag is cleared before leaving this function. - Modify this function so that it returns the read value (unsigned char I2C_READ_DATA(void)).
void INIT_I2C_IT(void);	This routine configures the interrupts after each I2C End of Transmission using interrupt channel INTB0. Don't forget to initialize your start-up file correctly.
void Enable_I2C_IT(void);	Enable I2C interrupts on channel INTB0.
void INTI2C_EndTrans(void);	**Interrupt subroutine** which occurs after an I2C end of transmission. The "user part" presents the way to deal with read data from the I2C bus. (it reads 10 bytes from EEPROM and stocks them in EEPROM_REC).

4.8.2.2.3 MFT files without DMA

The MFT function file is <mft\appli1\mft.c> and contains 6 functions initialized to generate two PWM signals using the two output pins of the MFT0.

Compare0 and Compare1 events are managed by interrupts.

Input A is used as a gate: as long as a +5V level is applied on it, the counter stops down-counting and the PWM signals are thus not generated.

The ports used for this application are:

– P4.6 T0OUTB

– P4.7 T0OUTA

– P7.3 T0INA

APPLI1\MFT.C functions

4.8.2.2.4 MFT files with DMA

The MFT function file is <mft\appli2\mft.c> and contains 7 functions initialized to generate a PWM signal using the MFT and a DMA channel.

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_MFT(void);	MFT initialization. It initializes the value loaded into the prescaler and into the counter.
void START_MFT(void);	Start the MFT.
void STOP_MFT (void);	Stop the MFT.
void INIT_MFT_IT(void);	This function configures the IT for the MFT.
void Enable_MFTCM_IT(void);	Enables Compare0 and Compare1 IT.
void INTMFT_Compare(void);	**Interrupt subroutine** which occurs after: - Compare 0 - Compare 1 Only one IT vector.

The port used for this application is:

– P9.3 T0OUTA

APPLI2\MFT.C functions

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_MFT(void);	MFT initialization. It initializes the value loaded into the prescaler and into the counter.
void START_MFT(void);	Start the MFT.
void STOP_MFT (void);	Stop the MFT.
void INIT_MFT_ITDMA(void);	This function configures the IT and the DMA for the MFT.
void Enable_MFTCP0_DMA (unsigned int * CompBuffer, unsigned int Count);	Enable Compare0 DMA.
void INTMFT_CompEOB(void);	**Interrupt subroutine** which occurs after a compare0 DMA end of block. The DMA is not re-initialized so only one block is transferred. Same vector as compare0 interrupt.
void INTMFT_OUF(void);	**Interrupt subroutine** which occurs after an underflow. It stops the timer when the whole PWM signal has been generated.

4.8.2.2.5 MFT files with DMA in swap mode

The MFT function file is <mft\appli3\mft.c> and contains 6 functions initialized to generate a PWM signal using the MFT and a DMA channel working in swap mode.

The port used for this application is:

– P9.3 T0OUTA

APPLI3\MFT.C functions

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_MFT(void);	MFT initialization. It initializes the value loaded into the prescaler and into the counter.
void START_MFT(void);	Start the MFT in the swap mode.
void STOP_MFT (void);	Stop the MFT.
void INIT_MFT_ITDMA(void);	This function configures the IT and the DMA for the MFT.
void Enable_MFTCP0_DMA(unsigned int * CompBuffer, unsigned int Count)	Enable Compare0 DMA.
void INTMFT_CompEOB(void);	**Interrupt subroutine** which occurs after a compare0 DMA end of block. The DMA is swapped.

4.8.2.2.6 RCCU file

The RCCU file is <rrcu\rccu.c> and contains 5 functions for controlling the RCCU in your application. The header file has no uninitialized constants since the functions are dedicated.

RCCU.C functions

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_PLL(void);	Initialize the PLL, Mul. by 6, div. by 1. Wait 500µs to stabilize the PLL.
void INIT_clock2(void);	INTCLK = CLOCK2 = EXT OSCILLATOR / 2.
void INIT_clock2_16(void);	INTCLK = CLOCK2/16 = EXT OSCILLATOR / 32.
void SWITCH_TO_EXTCLK(void);	Stop the Xtal oscillator and select the external clock (if present).
void BACK_TO_XTAL(void);	Restart the Xtal oscillator and select it as the clock source.

4.8.2.2.7 SCI files

The SCI function file is <sci\appli1\sci.c> and contains 4 functions initialized to send ten bytes from the ST9+ to the serial link (RS232).

It uses End of Transmission interrupts.

Port used:

– P9.4 Tx (S0OUT)

SCI.C functions

4.8.2.2.8 SCI files using DMA

The SCI function file is <sci\appli2\sci.c> and contains 4 functions initialized to send 26 bytes from the ST9+ to the serial link (RS232) by using the DMA.

It uses DMA channel (from memory).

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_SCI(void);	Initialization of general parameters dedicated to the SCI: - Baud Rate Generator: 9600 baud. - Characters: Eight data bits, one stop bit, odd parity. - Pins... The baud rate generator must be initialized following this structure (BRGHR first and BRGLR at the end)
void INIT_SCI_IT(void);	Initialization of the IT: vector, priority.
void SCI_SendByte(unsigned char ToSend);	It sends byte ToSend to serial link. This routine also enables end of transmission IT.
void INTSCI_TransmitReady(void);	**Interrupt subroutine** which occurs after the transmission of one character. If 10 bytes have been transferred, it disables the IT and so stops the transmission.

Port used:

– P9.4 Tx (S0OUT) (Uses End of Transmission interrupts).

SCI.C functions

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_SCI(void);	Initialization of general parameters dedicated to the SCI: - Baud Rate Generator: 9600 baud - Characters: Eight data bits, one stop bit, odd parity. - Pins... The baud rate generator must be initialized following this structure (BRGHR first and BRGLR at the end)
void INIT_SCI_ITDMA(void);	Initialization of the IT: vector, priority. Initialization of DMA in transmission
void START_SendDMA(unsigned char * TransmitBufferMem, unsigned int SCI_count);	After a few extra initialization, the transmission starts by loading data register(TXBR) with the first value to send.
void INTSCI_TransmitEOB(void);	**Interrupt subroutine** which occurs when a whole block of data has been transferred using DMA. In fact, the buffer ”table” is sent three times and then DMA is disabled.

4.8.2.2.9 SCI files using DMA in the loopback mode

The SCI function file is <sci\appli3\sci.c> and contains 5 functions initialized to send 26 bytes from memory to serial link.

The SCI is in loopback mode, so that the bytes sent are then received by the SCI and stored in the register file (from R100).

It uses DMA channels:

– From memory to peripheral for the transmission

– From the peripheral to the register file to receive the bytes.

No ports are used.

SCI.C functions

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_SCI(void);	Initialization of general parameters dedicated to the SCI: - Baud Rate Generator: 9600 baud - Characters: Eight data bits, one stop bit, odd parity. - Pins... The baud rate generator must be initialized following this structure (BRGHR first and BRGLR at the end)
void INIT_SCI_ITDMA(void);	Initialization of the IT: vector, priority. Initialization of DMA in transmission and reception. Bit b0 of DAPR registers must be set to 1!!!
void START_DMA(unsigned char * TransmitBufferMem, unsigned int SCI_count)	After a few extra initialization, the transmission starts by loading data register(TXBR) with the first value to send. DMA is enabled in transmission and reception
void INTSCI_TransmitEOB(void) ;	**Interrupt subroutine** which occurs when a whole block of data has been transferred using DMA. Only one block of data is sent (DMA not re-initialized after this IT).
void INTSCI_ReceiveEOB(void);	**Interrupt subroutine** which occurs when a whole block of data has been received using DMA. Only one block of data is received (DMA not re-initialized after this IT

4.8.2.2.10 SCI files using character matching

The SCI function file is <sci\appli4\sci.c> and contains 5 functions initialized to receive data on the input pin of the SCI and it only reacts when it receives character SCI_MATCH (defined in sci.h).

Port used: P9.5 Rx (S0IN)

SCI.C functions

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_SCI(void);	Initialization of general parameters dedicated to the SCI: - Baud Rate Generator: 9600 baud - Characters: Eight data bits, one stop bit, odd parity. - Pins... The baud rate generator must be initialized following this structure (BRGHR first and BRGLR at the end)

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_SCI_IT(void);	Initialization of the IT: vector, priority. The only interrupt which is enabled is ”character match”
void INTSCI_ReceiveMatch(void);	This **interrupt subroutine** occurs when the SCI cell has received the matching character. In this example, it displays 0 on a 7-segment LED connected to port 4.

4.8.2.2.11 ST files

The ST function file is <st\timer.c> and contains 6 functions initialized to generate a simple PWM signal using the Standard Timer (programmable duty cycle).

Port used:

– P4.3 STDOUT

– P9.5 Rx (S0IN)

TIMER.C functions

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_ST(void);	Standard Timer initialization. It must be used before any use of the ST. It initializes the value loaded into the prescaler and into the counter To use the ST in output mode, you **must** initialize the output pin in push-pull, **alternate function**. This means (PXC0R,PXC1R,PXC2R) = (1,1,0).
void START_ST(void);	Start the Standard Timer.
void STOP_ST (void);	It stops the timer counter
void INIT_ST_IT(void);	This routine configure the interrupts after each End of Count using interrupt channel INTA4. Don’t forget to initialize your start-up file correctly.
void Enable_ST_IT(void);	Enable Standard Timer Interrupt.
void INTST_EndCount(void);	**Interrupt subroutine** which occurs after an end of count

4.8.2.2.12 WDT files to generate PWM

The WDT function file is <wdt\appli1\wdt.c> and contains 7 functions initialized to generate a simple PWM signal using the WDT (programmable duty cycle).

Port used:

– P4.4 WDOUT

WDT.C functions

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_WDT(void);	WDT initialization. It must be used before any use of the WDT. It initializes the value loaded into the prescaler and into the counter To use the WDT in output mode, you**must** initialize the output pin in push pull, **alternate function**. This means (PXC0R,PXC1R,PXC2R) = (1,1,0)
void START_WDT(void);	Start the Watchdog Timer.
void STOP_WDT (void);	It stops the counting of the timer (useless in watchdog mode).
void Restart_Watchdog(void);	This routine refreshes the Watchdog counter: At each End of Count, if this function has not been used before, a reset is generated internally. The periodic use of this function is the only way to avoid a reset.
void INIT_WDT_IT(void);	This routine configure the interrupts after each End of Count using interrupt channel INTA0. NMI is thus configured as the Top Level Interrupt. Don't forget to initialize your start-up file correctly.
void Enable_WDT_IT(void)	Enable the WDT interrupt.
void INTWDT_EndCount(void);	**Interrupt subroutine** which occurs after an end of count. The code written in the user part allow the generation of a simple PWM signal (programmable duty cycle).

4.8.2.2.13 WDT files in watchdog mode

The WDT function file is <wdt\appli2\wdt.c> and contains 4 functions initialized to enable the WDT in watchdog mode and refreshes it regularly.

No port used.

WDT.C

FUNCTION	DESCRIPTION AND COMMENTS
void INIT_WDT(void);	WDT initialization. It must be used before any use of the WDT. It initializes the value loaded into the prescaler and into the counter.
void START_WDT(void);	Start the Watchdog Timer.
void STOP_WDT (void);	It stops the counting of the timer (useless in watchdog mode).
void Restart_Watchdog(void);	This routine refreshes the Watchdog counter: At each End of Count, if this function has not been used before, a reset is generated internally. The periodic use of this function is the only way to avoid a reset.

5 USING THE DEVELOPMENT TOOLS

5.1 DEVELOPING IN C LANGUAGE

Although the ST9+ C compiler is an optional product, you are strongly advised to write your software using a High Level Language. Naturally for the sake of optimization, especially to get the best execution times from certain frequently-used pieces of code, assembly language will still remain the right choice. This will be true at least for the initialisation file. But writing a complete program in assembler has few advantages and many drawbacks, so there can be no economic justification for using only assembler.

This is why some of the examples in this guide are written in assembler and some in C. The development tools allow you to mix both languages easily.

A useful book, if you have some experience of C language is ”The C language”, by Kernighan and Ritchie, second edition.

5.2 AVAILABLE TOOLS

The tools available for the ST9+ family are the following:

– A Software development package, named GNU9P, that includes a C-compiler and a make utility

– A range of emulators, for the various sub-families (ST90158 or ST90135 etc.)

You have two options in terms of development tool products. A software development package is delivered with each emulator. This does not include the C-compiler. You must purchase it separately. So with the emulator package alone you can only do assembler programming.

As has been said you are strongly advised to use C programming as much as possible, for the obvious reasons of structuring the source code, of code portability and reusability. In addition, developing the program is much easier and more reliable because from the controls and checks done by the C-compiler.

However the case where a program require very fast processing, the C language is impractical so that they must be carefully written in assembly language

Another point worth mentioning is the development environment, which is the part of the software that deals with the production and the modification of the source files.

In the past a straightforward text editor has been used to edit the source code. Most developers now use one of the so-called Integrated Development Environments (of which the most popular is probably the range of Borland’s Turbo- products). These environments have a multiple window text editor, a facility to generate make files and to call the compiler, processing of compiler error messages and so on. These facilities are not part of the GNU9 tools.

This chapter addresses this issue, and offers solutions to give you the same ease of use as in other development packages.

Note: The Companion software includes a powerful text editor and all the files that make up the various examples. For easy installation we suggest you keep exactly the same directory structure, by copying the whole contents of the downloadable file (using Windows File Manager's drag-and-drop feature) to a directory created with the name of your choice.

5.3 REQUIRED CONFIGURATION

The required configuration is a PC equipped with a 386 processor or better with Windows 3.11 or 95 and either the Assembly language ST9+ Software Chain, or the ST9+ GNU C Chain.

5.4 INTRODUCING THE DEVELOPMENT TOOLS

The programming tools available for the ST9+ are known as GNU-9 tools. They include a C-compiler, a macro-assembler, a linker, a make utility and two different debugging hardware systems, both with the same user interface running under Windows.

The GNU-9 is a set of MS-DOS programs that can be driven from a single program called GCC9. This program is capable of calling the following programs in turn:

Main block name	File name	Block name	Action
C-compiler	cpp9	C pre-processor	Expands the macros, inserts the include files, removes the disabled conditional compilation blocks.
	cc9	C-compiler	Translates C-code into assembly source text.
Macro-assembler	tr9	Assembler pre-processor	Expands the macros, inserts the include files, removes the disabled conditional compilation blocks.
	gas9	Assembler	Translates assembly language into machine code.
Linker	ld9	Linker	Links the different object files, positions the code at predefined addresses in memory.

You can start the GCC9 using command-line options that indicate which of the blocks shown above will be called. Thus a single command, GCC9, can compile, assemble, and link, separately or in sequence. However, GCC9 cannot handle several files at a time, as is the case in nearly all applications. For this reason you can use another utility to drive the tools, gmake. Gmake is a make utility that allows you to define the structure of the program to be created (list of file names, include files, their dependencies) in two program description files. Gmake checks for the dates of the files and automatically starts the compiler, the assembler, and the linker only for those files that have been changed since last time gmake was invoked.

5.5 SOFTWARE ENVIRONMENT

The software tools consist of:

– The development chain, called GNU9P, and

– The debuggers of various types depending on the emulator hardware used.

ST9+ 4.2 toolchain runs on MSDOS 6.2, WIN-3.1, WIN-95, WIN-NT platforms, SUNOS 4.1.x and SOLARIS 2.0 platforms.

5.5.1 DOS Environment

Under this environment, you can only possible to write and compile the program. You can use any text editor, like Edit (delivered with MS-DOS 6). However, a more efficient editor is advisable, like Codewright.

All components of 4.2 PC-hosted toolchain obey DPMI 1.0 standard, a.k.a. “DOS Protected Mode Interface”; as a consequence, they run faster, use less memory, have far less memory exhaustion problems and do not cause DOS file name extension incompatibilities or memory configuration intricacies.

5.5.2 Windows Environment

Windows is the most convenient environment for handling the complete development cycle. We have suggested two editing environments here, but other editors would suit as well. The compiler, although it runs under DOS, can be executed directly from the editor, or using an icon in program manager.

5.5.2.1 Programmer’s File Editor

The Programmer’s File Editor is a Multi-Document Interface editor that has practically all the features a program developer could imagine. It is public-domain software and is supplied in the Companion software with the author’s permission. It allows you to edit as many files as you want simultaneously, and offers search and replace, line numbering, drag-and-drop editing, and many more features. For example you can define different page layouts for each type of file (C source text, assembler source text, etc.), or launch an external application, such as gmake or GCC9, and collect the results (the error report) in another editor window. This helps locating the lines in the files where errors have been detected. In addition, it can be connected to Windows-style help files, for context-sensitive help similar to that found in commercial development environments.

5.5.2.2 The Turbo C++ Environment

If you already have Borland’s C++ Integrated Development Environment, it makes sense to use it as the file editor to write the ST9+ programs. This environment has very useful features, including syntax highlighting that shows the various elements used in a C source text (identifiers, numbers, reserved words, comments, etc.) in different colours. It has a very good on-line help with information on usage and syntax of all words defined in both ”classical” C (Kernigan and Ritchie) style and ANSI style.

This environment naturally has its standard compiler for producing DOS executable files. This allows you to develop ST9+ applications by first writing and debugging the parts of the program that are not too microcontroller-oriented such as data processing, standard input-output, etc. without having to actually use the circuit board prototype and without using the specific ST9+ emulator. In some cases, you could develop the whole ST9+ application by simulating the ST9+ features using specially written C functions to handle the low-level aspects of the program. These can be replaced later by the correct ST9+ routines when it is time to use the real ST9+-based circuit board.

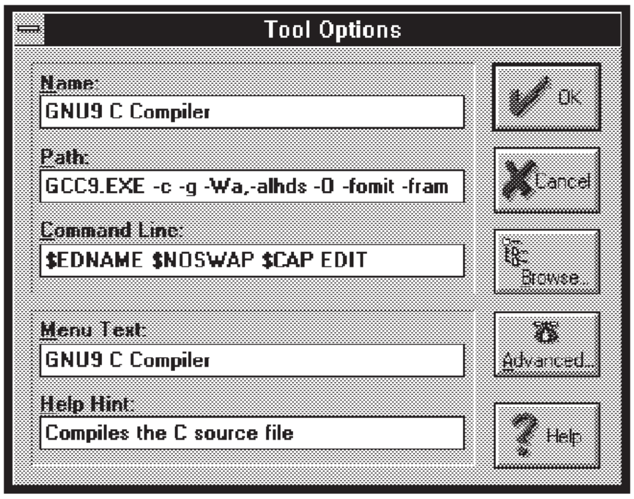
As a final note, you can easily add two new options to the tool menu: ”GNU9 C Compiler” and ”Make GNU9”, to launch the GCC9 or the gmake utility in a concealed DOS box. You can monitor the progress by opening the corresponding listing file.

To add these new options, perform the following steps:

In the ”Option” menu, select ”Tools”. The list of the available tools appears.

To add a new tool, press the ”Edit” button. A dialog box appears showing the settings of the tool that was highlighted in the list. Changing the name will automatically create a new tool.

Define the new tool by changing the various fields of the dialog box so it looks like the following:



Name:
GNU9 C Compiler

Path:
GCC9.EXE -c -g -Wa,-alhds -O -fomit -fram

Command Line:
$EDNAME $NOSWAP $CAP EDIT

Menu Text:
GNU9 C Compiler

Help Hint:
Compiles the C source file

The Path field contains the invocation line with the arguments. Here, the field is too small to show all the arguments. The complete line is:

```
GCC9.EXE -c -g -Wa,-alhds -O -fomit-frame-pointer -Wall -mparmusp
```

The Command Line field includes three macros whose meaning is explained in the Turbo-C++ on-line help. The macros used here are:

$EDNAME	Appends the name of the currently active window to the invocation line, so the compiler will take it as the source file.
$NOSWAP	Prevents the DOS box running the compiler from displaying in foreground.
$CAP EDIT	Captures the results of the compilation and on termination opens a Capture Window in the integrated environment showing all the text produced at the compilation phase.

Pressing OK will close the window and include the new tool in the Tool menu. This tool is saved and is available for later editing sessions.

You can use the same procedure to install the Make tool. The same steps should be repeated, with the proper parameters in the fields.

5.6 INSTALLING THE PROGRAMMER'S FILE EDITOR

The PFE is available in directory \PFE of the companion software. It does not have an installation program. Follow this simple procedure:

Copy pfe.exe, pfe.hlp, $pfedos.exe and $pfedos.pif to the directory of your choice. It's a good idea to call it PFE. It is not necessary that it be in the Path, but it is important that all the files be in the same directory.

In Program Manager, add the PFE icon. To do this,

Select the group where you want to put the PFE icon

Select the File/New menu option. In the popup window select New Program.

In the property window press Browse, then select the file PFE.EXE and press OK. If desired, fill in the ”program name” and ”default directory” fields and press OK.

To start with a context that fits the explanations in this book, the PFE.INI file is added in the same directory though it is not a part of the PFE product. Just copy this file to the Windows directory to find PFE already configured with the Help files, the DOS commands and a few Most Recently Used files. They may not suit the your own configuration, but can serve as an example to help you to configure it for your own machine.

The PFE icon is added to the desired group. To start PFE, just double-click on it.

It is not the aim of this book to explain how to use PFE. You can use it just as a basic text editor like Textpad. PFE is delivered with several text files that give basic information on the product. In addition, a plain Windows-style context-sensitive Help is available. We suggest you take a few minutes to exploring the capabilities of PFE that are really enjoyable. In particular, you should pay attention to the following functions:

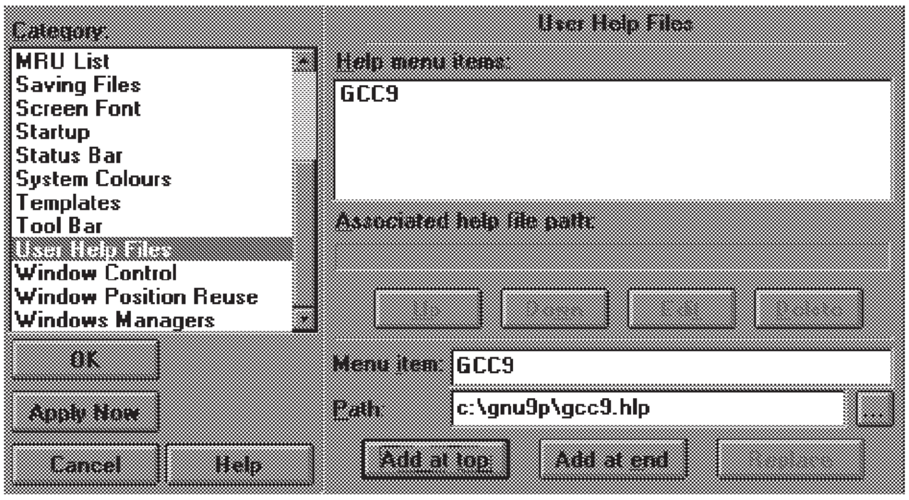
In the Options menu, the Default modes and Current modes commands allow you to define the properties of the source text according to the language used: assembler or C. For each of these, using the Edit Modes button, you can set different options including tabulation, end of line wrapping, and more. Each setting group can be associated with a file extension so that opening a source file will automatically format it the right way.

The Execute menu allows you to start other applications either under DOS or under Windows, and to pass parameters to them such as the name of the currently active source file. These commands and all other settings are stored in the PFE.INI file in the Windows directory, so you can reuse the same commands again from one editing session to the next one without having to retype them.

Another very powerful feature is that the report generated by the application launched this way is copied in another text window that opens automatically when the external process (compilation, etc.) is finished. The error report gives the line numbers where the errors were found. You only have to move the cursor to the corresponding line to correct the mistake.

The GNU9 package includes a set of help files in the GNU9\HELP directory. At source level, the most useful file is GCC9.HLP. If provides help both for the compiler, assembler and the linker, as well as for the C and assembly language syntax. PFE allows you to use this file in context-sensitive mode, which is extremely convenient.

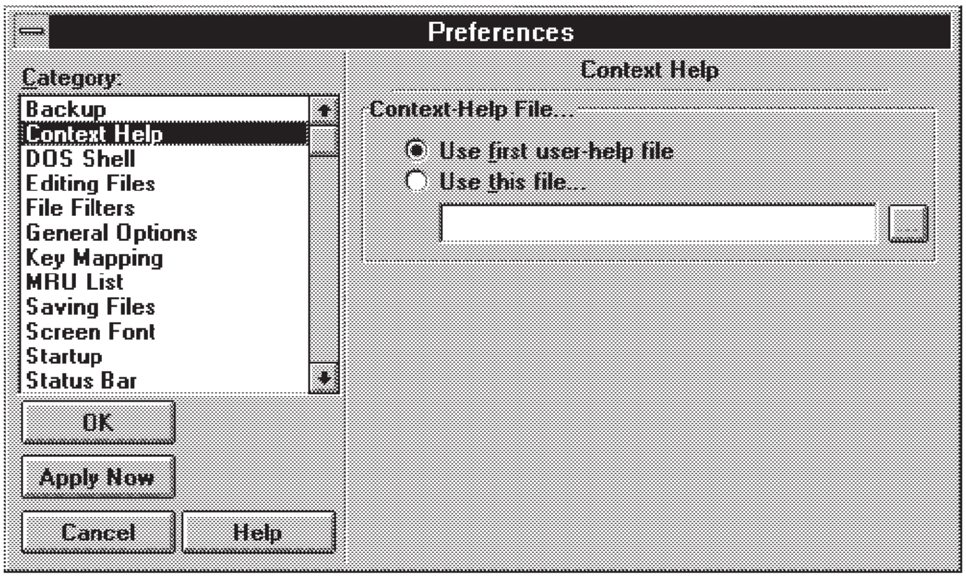
To connect the help files to PFE, select Options/Preferences and choose User Help Files in the list. In the Menu Item field, type GCC9. Press the browse button at the right of the Path field (this button is identified by ”...”). Navigate through the directories to find the file GNU9P\GCC9.HLP. The Preference box will look like this:


Category:
MRU List
Saving Files
Screen Font
Startup
Status Bar
System Colours
Templates
Tool Bar
User Help Files
Window Control
Window Position Reuse
Windows Managers

User Help Files
Help menu items:
GCC9
Associated help file path:
Menu item: GCC9
Path: c:\gnu9p\gcc9.hlp

Press Apply Now, then OK. This adds item GCC9 to the Help menu the bottom of the list. More files can be added to the list by repeating the same process.

To make PFE use this file as the context-sensitive help, select Options/Preferences again, and select «Context Help» in the list. Then by selecting ”Use the first user-help file”, the first file in the supplementary list of the Help menu will be used.



Context Help: [✓ Use first user-help file] Use this file...

Press Apply now, then OK.

To check the operation of the context-sensitive help, open a C source file or type a few C statements. Put the cursor somewhere within a word for which you want help (e.g. if). Select Help/Context help or simply press <CTRL>F1. The help selection box appears the usual way.

All these features make PFE is a very useful and easy to use environment for developing ST9+ software.

An example of a PFE.INI file is given under the name GNUPFE.INI. It includes the commands for assembling a file and calling the make utility.

5.7 ASSEMBLER, LINKER AND HEX FILE CONVERTER

5.7.1 Description

The assembler, linker, librarian and hex. file converter suite is supplied with the emulator. The assembler allows you to use the full capabilities of the ST9+. It includes a powerful macro language that can give the application code a hint of high-level language. However, developing a large program in assembler is impractical, and, unless you have extreme requirements in terms of optimization, we strongly advise you to use C-language to as wide an extent as possible.

Though high level language is a safer and easier way to write programs, some time-critical applications must be written in assembler to get the most out of the ST9+'s power. In such cases, use the registers for as many variables as possible, even if there is RAM available, and choose the instructions you use for their execution times. Note that all addressing modes are not available for all instructions. This implies you must carefully choose the structure of the data in the registers. This is particularly true for the working registers. There are many ways to use the working registers: a single group of 16 registers, or two groups of registers. To access values in registers, you can use either direct access, or change the working register set each time another group of data is to be handled.

The result of these choices is not self-evident. It is sometimes necessary to write two or more variants of the same code and calculate the execution times using for example a spreadsheet program.

Notes: The GNU assembler and linker, unlike most other similar products, do distinguish uppercase and lowercase letters. So the same care must be taken in assembler as in C-language regarding case.

As in any assembler, symbols that must be exported to other modules must be declared global. However, the corresponding .extern statement is merely ignored by the assembler. At assembly time, all unknown symbols are considered external. It is only at link time that the error is signalled, with the name of the source file and the line number where the error occurs.

The set of .INC files that describe the internal structure of each type of ST9+ variant, each contains a set of include statements to more include files that describe each peripheral. ST90158 for example, the file ST90158.INC, reads as follows:

```
; Include file for the definition of the registers and bits for the
; ST90158
.nlist
.include  ”c:\\ST9p\\inc\\system.inc”  ; System register
.include  ”c:\\ST9p\\inc\\page_0.inc”  ; Page 0 register
.include  ”c:\\ST9p\\inc\\eeprom.inc”  ; EEPROM register
.include  ”C:\\ST9p\\inc\\sec_reg.inc” ; Security register
.include  ”c:\\ST9p\\inc\\io_port.inc” ; I/O port register
.include  ”c:\\ST9p\\inc\\mftimer.inc” ; MF Timer register
.include  ”c:\\ST9p\\inc\\ad_c.inc”   ; A/D converter register
.include  ”c:\\ST9p\\inc\\sci.inc”    ; SCI register
```

```
.include  ”c:\\ST9p\\inc\\mmu.inc”   ; MMU register
.list
```

The paths mentioned here are absolute. This means that if the GNU9 is installed in the default directory, i.e. C:\GNU9P, everything works well. If, however, the installation directory is different, the include statements will fail. You them have to change this file to read as follows:

```
; Include file for the definition of the registers and bits for the
; ST90158 family
.nlist
.include  ”system.inc”  ; System register
.include  ”page_0.inc”  ; Page 0 register
.include  ”eeprom.inc”  ; EEPROM register
.include  ”sec_reg.inc” ; Security register
.include  ”io_port.inc” ; I/O port register
.include  ”mftimer.inc” ; MF Timer register
.include  ”ad_c.inc”   ; A/D converter register
.include  ”sci.inc”   ; SCI register
.include  ”mmu.inc”   ; MMU register
.list
```

The assembler and the linker can be invoked in sequence using the GCC9 command. This command is easy, but not practical in most cases, since it is limited to single-file programs. If your program is made of several source files, use of the make utility. This allows you to fine tune the memory addresses, and to build an automatic assembling and linking process that, once defined, is a very handy feature, for there are few programs that are assembled only once!

The hex. file generator is a program that converts the linked object into an absolute file coded in Intel Hex. format. Most PROM programmers accept this format. It is not called within GCC9, and it is a good idea to include the following command line in the make file:

```
Intel9 <name of the object file> > <name of the hex file>
```

This ensures that an up-to-date version is always available for programming the ST9+ on-chip PROM.

5.7.2 Installation and Configuration

Installing the software tools is very simple. Insert the first diskette of the pack in the 3.5” drive, start the set-up program from the diskette, and specify the installation directory in the dialog box. The installation then starts.

To be able to run the GCC9 package, a DOS icon must include the call to the Setgnu9.bat file with the path of toolchain directory to initialise the DOS environment. This icon (ST9+ 4.2 Toolchain session) is automatically generated during the GNU Toolchain installation. Here's the Setgnu9.bat content:

```
@echo off
if x%1==x goto noarg
if x%2==x goto defver
set GCC9_VER=%2
:defver
set GCC9_VER=4.2
:setting
set TMP=c:\TEMP
set GCC9INC=%1\include
set GCC9_EXEC_PREFIX=%1\bin\
set PATH=%1\bin;%PATH%
goto end
:noarg
echo ”Usage: %0 <toolchain location> [ <version number> ]”
:end
```

Here are the lines used for this configuration. Since they mention file paths, they must be changed to match the directory structure of your computer. These lines are:

Command line	Use
set GCC9_VER=%2 or 4.2	This environment variable is set to the version used by the make to call the good executable.
set PATH=%1\bin; %PATH%	Extend the standard path to give the path for GCC9.exe (here, the path is c:\gnu9p\bin).
set GCC9_EXEC_PREFIX=%1\bin\	This environment variable is set to the path of GCC9 and all its components. It allows GCC9 to find them.
set GCC9INC=%1\include	This environment variable is set to the path of the include files. These are both .inc files for the assembler, and .h files for the C compiler, and they all define the register names of the selected variant of the ST9+.
set TMP=c:\TEMP	This environment variable is set to the path of the temporary working files. They can be put virtually anywhere, since they are destroyed after the process is finished. The only constraint is that there is enough disk capacity to accommodate them.

The path and the set initialized give you the possibility to compile your application by using the Execute DOS Command and Capture Output of the PFE editor. The command gmake will update the application and the compilation result is given in the CommandOutput1 PFE windows.

5.7.3 Using Registers in Assembler

Using registers for variables is always more efficient than using memory in terms of speed. In addition, the code size is likely to be smaller. However, this requires more care from the programmer, since you have to allocate the registers entirely by yourself.

In assembly language, the following statements[6]:

```
Data:.blkb 4; a double word of data
Index:.blkw 1; a pointer
Charact:.blkb 1; a character
```

make the assembler assign the value Data+4 to the label Index, and the value Index+2 to the label Charact. However, when using registers, there is no such mechanism that places the data in a row, taking into account the size of each item. It is up to you to define the register numbers yourself, avoiding overlapping other data with double or quadruple registers. Moreover, if the program requires inserting one more register in the middle of an ordered register set, all register numbers following the addition must be corrected by hand, with chances for making mistakes.

So, using the registers requires care when allocating them. Many programming errors come from incorrect assignment. In the applications described in this book, the register allocation process was done very early in the conception of the program. However, this did not prevent us from having to re-allocate them once we started writing the source code.

When using word or double word registers, you must take care that the word-wide instructions (the ones which mnemonics ending with the letter w, such as ldw) must always start on an even number. Taking advantage of these instructions sometimes leads to holes in the register map.

5.8 C COMPILER

The GNU9 C-compiler allows you to program the ST9+ using either the traditional style of C (Kernighan and Ritchie), or the ANSI-standardised C language. All the examples of this book are written in ANSI dialect.

As we will see later when discussing the linker and the program initialisation module, the C language is not aware of RAM, ROM and peripherals. For this reason, the GNU9 C-compiler includes non-portable extensions that allow it to handle the ST9+-specific programming as-

[6] ST9+ Family GNU Software tools V2.0, first part, § 4.2.2 and 4.2.3; pseudo instructions **.blkb** et **.blkw**.

pects. Since we assume you have a general knowledge of the C-language already, only the GNU9 C-compiler specifics will be discussed here.

The main ST9+ C-language issues are:

Adding assembler statements in C source text

Accessing the register file

Memory allocation of variables

Compiler command-line options

5.8.1 Registers vs. Memory and Register Allocation

The registers of the ST9+ are numerous, and can serve many purposes. This means we have to define in detail how they will be used. There are two cases that use the registers differently.

5.8.1.1 MCU without external RAM

With the ST9+ you can write a program that uses C language. You must still be careful because the standard C predefined functions use up a variable amount of RAM that in some cases can exceed the whole RAM available in the chip.

5.8.1.2 The ST9+ Chip has External RAM Added

When plenty of memory is available, there is no question that you should use C language. When you have to process large amounts of data, it is almost impossible to handle them using assembly language. Then, the registers will probably be under-used, since one register group will be used for the main program, and possibly another group for each interrupt level (there are seven), and the system and paged register groups. The stack will necessarily reside in RAM, at least for parameters and return values. You can specify that the compiler uses two stacks, one in memory for arguments, the other in registers for return addresses, so the register file will be neatly involved in the program layout, giving a good compromise. This leaves room for further optimization. If a given routine is called often, you can speed up execution by rewriting it in assembly language and using registers instead of memory for the local variables, since accessing registers is faster than accessing memory.

5.8.2 Using Assembler Statements in C Source

You need to use assembly language when you comes to the point of driving the most low-level features of the ST9+, such as setting a register, enabling and disabling interrupts, switching register pages, etc.

The assembly language available within the C compiler allows you to include assembly statements within C source text. You can even write the whole body of a C function in assembler, writing only the function declaration in C. To do this, you use a sophisticated syntax to pass the C variables to the assembler code and the return value calculated in assembler back to the C function. Refer to the GNU C Compiler manual.

You can also write an assembler source text that provides global functions that are called from a C function. However, most of the uses of in-line assembler statements within a C-function are for simple actions, like enabling interrupts. When the code is only a few instructions long, you can define it as a macro:

```
#define DI() asm ( ”di”) ;/* inhibit interrupts */
#define EI() asm ( ”ei”) ;/* enable interrupts */
```

To be syntactically correct, the macro that includes only the assembler statement di (or ei) is given a name that sounds like a C function: DI() (or EI()). This allows you to include these statements in a C source text that still looks like a regular C text.

Here are other examples. First, a function to push current page number on the stack, and another one to pop the previous page number off the stack:

```
#define PushCurrentPage() asm ( ”push R234” ) ; /* Save page pointer register */
#define PopCurrentPage() asm ( ”pop R234” ) ; /* Save page pointer register */
```

Now, an example of a macro that changes the current page number:

```
#define SelectPage( Page ) asm ( ”spp %0”:: ”i” (Page) ) ; /* select page
timer */
```

The previous example uses one of the available features for defining the way parameters are handled. Here, %0 means the first (and only) parameter passed when calling the macro. ”i” indicates that it should be a constant, and Page is the formal name of the parameter.

Here is an example of the use of these macros:

```
void CharacterMatch ( void )
   {
   EI() ;   /* enable high-level interrupts */
   PushCurrentPage() ;
   SelectPage( SCI1_PG ) ;
   S_ISR = 0 ;/* Reset all */
   ....
   PopCurrentPage() ;
   }
```

5.8.3 Accessing the Register File

Another very common use of assembler statements involves the register file. Since no actual C statement gives access to data located in registers, writing an assembler statement is the only way to do it. So all ST9+ programs include a few assembler statements. To access a register, you can use the following very convenient notation:

```
register unsigned char Events asm(”R59”) ;
register long Position asm(”RR32”) ;
```

In this example, we define two C variables, one of type char, i.e. a single byte, that is associated with register R59 ; the second one is of type long, and is associated with the two register pairs RR32 and RR34. The advantage of this notation is that it only has to occur once in a source file. All subsequent uses of these registers use the C identifier Events or Position in regular C statements and expressions. These declarations are not global. They must be repeated in all the source files that use them. Thus, it is advisable to include them in the header file (.h) that is included in some or all of the C source files.

Here is an invocation example:

```
   Shift = ( Position & 0x7f ) - ( Pos & 0x7f ) + Shift ;
```

The piece of code given in the previous paragraph also includes a register invocation, in the line:

```
   S_ISR = 0 ;/* Reset all */
```

where S_ISR is defined in the SCI.h file that belongs to the GNU9 package as:

```
register volatile unsigned char S_ISR  asm(”R247”);  /* interrupt status
register*/
```

(In both assembly and C languages, include files are supplied that have symbolic names predefined for all the peripherals. These names are unique for each peripheral: several different names relate to the same register, but in a different page. Using the name of a register does not automatically select the proper page. You must precede this statement by another one that selects the appropriate page.

Declaring the register-based variables keeps the code tidy, since outside the declaration part, there is no indication that the variable is allocated to a register. However, this syntax has limitations:

You cannot define a pointer to a register variable. Hence, the * and & operators are not allowed

A register variable cannot be the basic type of an array or a structure member

5.8.4 Memory Allocation and Variable Qualifiers

As said earlier, standard C-language is not aware of the various address spaces. The only difference in the variables come from:

Size and structure: char, int, long, struct...

Scope and properties: static or not, extern...

These classes are defined in the ANSI standard and naturally exist in the GNU9 C-compiler, we will not deal with them here. However, there are other qualifiers that are used by the compiler for optimization. This is the topic of this section.

5.8.4.1 The Const Qualifier

The const qualifier simply defines that the variable is read-only. Any attempt to write to it will produce an error. This is useful for read-only registers, and for data used as constants, especially if constants are made with initialised variables, such as:

```
const float Pi = 3.14592 ;
```

The Pi variable in this case is allocated in the .data section and can be mapped in RAM during the link phase, and set to its initial value. We do not want it to change during program execution. Declaring it const does not protect it from being written in case of a software failure but it guarantees that nowhere in the C source code is it assigned any value.

5.8.4.2 The Volatile Qualifier

From the C point of view, all variables are only storage, which means that the value of a variable can only change when an assignment is made to it. Thus, if the compiler encounters the following statements:

```
   if ( Var == 0 )
          {
          ....       /* do something */
          }
   if (Var == 0 )
          {
          ...        /* do something else */
          }
   ...
```

it is likely to consider that if Var is zero the first time, it will be the second time, if no assignment is made to Var in the first block. Thus, the compiler may want to group the two braces in one, such as:

```
   if ( Var == 0 )
          {
                     {
                     .... /* do something */
                     }
                     {
                     ...  /* do something else */
                     }
          }
   ...
```

This may not do what you intended to do. For example, suppose Var represents the result of an Analog to Digital Converter whose interrupt service routine reads the result and copies it into the Var variable. The value read might well change while processing the first block. Especially if, for example the first block triggers another conversion. What we wanted can be defined as:

```
If the value is zero, then do another conversion,
and if the value is still zero, do something special.
```

What the compiler would do means:

```
If the value is zero, then do another conversion, and do something special.
```

Which is different and probably wrong.

To avoid such conditions, the volatile qualifier is available in the GNU9 C-compiler. If you write the following line in the variable declaration section of your source text:

```
volatile char Var ;
```

you can make sure that the compiler will not do the grouping shown above and keep the two separate IF statements.

Always add the volatile qualifier to a variable that is either real memory storage shared between several processes and/or interrupt service routines, or a variable representing an I/O register that can be changed by hardware. If a register is read-only (which means that it is only changed by hardware), it should have both qualifiers const and volatile.

5.8.5 Interrupt Service Routines

Interrupt service routines can readily be written in C. This is a valuable feature, since you do not have to bother with the interrupt prologue and epilogue. However, to be able to write working interrupt service routines you need to pay special attention to the following points.

An interrupt service routine can be any function in the form:

```
void InterruptServiceRoutine ( void ) ;
```

To tell the compiler that a function is an interrupt service routine, there are two ways:

Either the function is preceded with a statement such as:

```
#pragma interrupt (InterruptServiceRoutine)
```

There must be one such statement for each interrupt service routine.

Or the whole file can be compiled with the option -miret saying that all the functions of that file are interrupt service routines.

The first method has been used in the examples mentioned.

5.8.5.1 Interrupt Vectors

Interrupt vectors are by no means automatically generated by the C compiler. It is up to you to add them in the start-up code written in assembler. This requires fully understanding how interrupt vectoring is organised. Refer to Section 3.4.1 for an explanation of the vector system. Writing the start-up file is covered later in this chapter (Section 5.9.3). There must be one entry in the vector table for each interrupt service routine. You must also take care to initialise the vectors to some value that corresponds to interrupts that are likely to occur by accident (e.g. the parity error vector for the SCI). This vector must lead to a routine that either discards the interrupt request, or processes the event to attempt a recovery.

5.8.5.2 Context Saving

Context saving consists of saving the working registers. It is taken care of by the compiler. However, this can be done in three ways:

Using either only the -miret option or the statement:

```
#pragma interrupt(InterruptServiceRoutine)
```

The working registers are pushed on the stack. This method is easy and safe, allowing function re-entrancy. However, it takes time to push the registers on entry and pop them on exit, and it consumes a lot of stack space.

With either the -miret -msrp -mgrpxx option, or the statement

```
#pragma interrupt(InterruptServiceRoutine, xx)
```

where xx is the number of a register group. The working register group is changed on entry to the group specified and restored on exit. This method is faster, saves stack space, but makes the interrupt service routine non re-entrant. Thus care is required to avoid either recursion or re-entrancy. Consider the following example:

```
#pragma interrupt(InterruptServiceRoutine, 8)
void InterruptServiceRoutine ( void )
   {
   ei () ;
   /* do something */
   }
#pragma interrupt(AnotherServiceRoutine, 8)
void InterruptServiceRoutine ( void )
   {
   /* do something else */
   }
```

Two different service routines share the same register group. The first one has re-enabled the interrupts, so that it can be interrupted by the second one. If this occurs, the second one will interrupt the first one, use the registers of group 8, and return. The first interrupt service routine will then resume, but its working registers are altered and something wrong may happen.

Take care that two interrupt routines that use the same register group do not interrupt each other.

5.8.5.3 Handling Peripheral Events and Interrupt Requests

When a peripheral has triggered an interrupt, it is because its state has been affected by some condition or external event. It requests to be serviced, which means that in addition to feeding it with data, or withdrawing data from it, there may be some registers to take care of, some bits to set or reset. Failure to do this could either keep the request still pending, or block further interrupts. For example, the error condition of the SCI triggers an interrupt that is serviced the following way:

```
#pragma interrupt(ReceiveError, 8)
void ReceiveError ( void )
   {
   EI() ;   /* enable high-level interrupts */
   PushCurrentPage() ;
   SelectPage( SCI1_PG ) ;
   S_ISR = 0 ;/* Reset all errors */
   ReceiveDMAPointer = ReceiveBuffer ;
   ReceiveDMACounter = RECEIVE_BUFFER_LENGTH ;
   PopCurrentPage() ;
   }
```

The ISR register that contains the pending requests is cleared in response to the error condition. This re-instates the UART into its normal working condition. In some cases, reading the incoming data clears the interrupt request. In other cases, it does not, and a bit must be cleared in some register.

5.8.6 Compiler and Linker Command-Line Options

The GNU9 command-line has a large number of options. They are intended to permit complete control over the compilation process. However, only a few ones are of regular use, the others are for special cases and fine-tuning the resulting program. Since all options are described in the ST9+ Family GNU C Compiler manual, you should consult this manual for details and the release note 4.2 (ST9P_RLS.PDF) placed in the ST9+ GNU C Toolchain windows. The same text is also available in the GCC9.HLP file.

The options that are of constant use deal mainly with the system hardware architecture, especially the memory organisation. The following examples are only guidelines, but they will allow you to get your application up and running quickly.

The system can have a large amount of RAM that can hold all the stack data. If the RAM size is limited, you have to use the register file for the System Stack. As mentioned in the previous chapter, C program return addresses can be either in memory or in the register file. However data (function arguments, local variables and return values) must reside in RAM.

Compiler options have two parts: one is fixed, the second depends on the memory model chosen. Linker options depend essentially on the memory model.

There are two ways to build the program:

– If the program is made of a **single C source file**, it can be compiled and linked in sequence using the GCC9 program which is the front-end of the whole GNU tool chain.

– If the program is made of **several files**, you compile all the files first then you do the linking. The compiler can be invoked using the GCC9 command. The linker must be invoked using the LD9 command.

Depending on the above cases, the linking options are noted differently.

5.8.6.1 New option versus ST9old option

To keep the possibility to work with ST9old microcontrollers, all options available in previous ST9 toolchains are available. In particular, options that select one or two spaces (-samepd or -mpd). The scheme used to call internal components is the same, except for tr9 which is not called by default any more (see below). In particular, default is to use -mpd for compiling and -I and -i for linking. The option -mst9 asks the compiler to generate code for an ST9 microcontroller.

For more details about new options, see the ST9+ Gnu C Toolchain Release 4.2.

5.8.6.2 Typical compiler options.

The following compiler command line options are suggested:

```
-c -g -Wa,-alhds -O -fomit-frame-pointer -Wall
```

where the options mean:

Option	Meaning
-c	Output an object file for later linking.
-g	Include in the object file all necessary information for the debugger to allow it to use symbolic names.
-Wa,-alhds	Tells the gcc9 to transmit the -alhds option string to the assembler. These options make the assembler generate the most complete assembly listing file to help you find problems more easily.
-O	Optimise the C compilation. This provides the most efficient code, both in terms of speed and memory size.

-fomit-frame-pointer	One of the optimization parameters. Tells the compiler it can save a working register by not maintaining the stack frame pointer inside functions that do not need one.
-Wall	Output the warning messages that relate to most of the cases where the correctness of the source code is questionable, or may lead to inefficient code.

5.8.6.3 Typical Linker Options

The main linker options are:

5.8.6.3.1 Linker Options When Invoked Using the LD9 Command

```
-m -I
```

where the options mean:

Option	Meaning
-m	Output a map file with the extension .map.
-I (uppercase i)	Generate in the code section (named .text) the table of the initial values of the variables to copy them into the RAM at start-up. This is detailed below in the start-up file paragraph. By default, the initial values are not included in the .text section.

5.8.6.3.2 Linker Options When Invoked Using the GCC9 Command

If the linker is invoked through the GCC9 command, the main linker options are:

```
-Xmap -noI
```

where the options mean:

Option	Meaning
-Xmap	Output a map file with the extension .map.
-noI (uppercase i)	By default, in the code section (named .text) the linker generates the table of the initial values of the variables to copy them into the RAM at start-up. This option cancels the code generation that is, it has the reverse effect of the -I option when invoked using the LD9 command. See the details below in the start-up file paragraph.

5.8.6.3.3 Options Specifying Memory Allocation

The options suggested below are for different situations. These options come in addition to the options described in the previous two sections. Alterations to the start-up code are also mentioned.

5.8.6.3.4 Compiling and Linking using the GCC9 Invocation

Plenty of RAM available	Start-up code: initialise SSP (RR238) in RAM, and bit SSP in register MODER (R234) to zero.
Less RAM available: using the register files is preferred for return addresses	-samepd -mparmusp Start-up code: initialise SSP (RR238) in registers, USP (RR236) in RAM, bit SSP to 1 and bit USP to zero in register MODER (R234).

5.8.6.3.5 Compiling using the GCC9 invocation and Linking using the LD9 Invocation

Plenty of RAM available	Compiler: Linker: nothing Start-up code: initialise SSP (RR238) in RAM, and bit SSP in register MODER (R234) to zero.
Less RAM available: using the register files is preferred for return addresses	Compiler: -mparmusp Linker: nothing Start-up code: initialise SSP (RR238) in registers, USP (RR236) in RAM, bit SSP to 1 and bit USP to zero in register MODER (R234).

5.9 PROGRAM CONFIGURATION AND INITIALISATION

Most programs are divided into several source files. This makes modifications easier and recompilation quicker, using the make utility that recompiles only the files that have changed since the last compilation.

How you split the source text into files is a matter of taste. However, a few rules are worth following for keeping things well organised. They are:

For sake of portability, assembler source text must not be mixed with C source text in the same file. In actual practice, assembly statements may be written in C, but reduce these to a minimum, e.g. to define variables in registers.

Code that is very low-level is very machine-dependent. This code should be in one or more specific files.

One or several higher levels of code can be defined according to the application. This is up to you, but in all cases, machine-specific code, even written in C, should not reside in the same file as a higher-level code. This allows changing the hardware configuration (assignments of the port pins, etc.) by changing only those files that are involved with low-level routines.

Once this layout is established, you can then write the files mentioned below. These files are used to configure the tools, so as that they know the program structure and can maintain it by compiling the right portions whenever they change. So you absolutely have to go through these steps even though they are not directly related to writing code. They are:

Writing the makefile

Writing the linker command file

Writing the start-up file

These files are specific to each project and must be added to the list of source files. Examples are given in the applications described in this book, and they are available in the companion software.

5.9.1 Writing the Makefile

The makefile is the template that puts all the program pieces together. It describes the modules interdependencies, allowing the gmake utility to build the program efficiently by processing only those source files that have been modified since the last invocation of gmake(7).

The make language is very powerful, for a make utility can be applied to almost any kind of processing where several files provide a single result. It is a generic machine that manages updates smartly. When developing programs for the ST9+, the makefile description is similar

(7) See GNU9P Make Utility Manual.

from one project to another. Most of the changes relate to the name and the number of the files involved. Thus, we propose here a skeleton makefile and we explain how you can tailor it for a specific application, without going into the advanced features gmake offers.

To run gmake, just invoke it by typing under MS-DOS:

```
gmake<Enter>
```

When no makefile is given, as in this example, gmake uses the file named ”makefile” or ”makefile.gnu”. If you want to use a different name, you must use this syntax:

```
gmake -f Mymake.mkf<Enter>
```

where Mymake.mkf is just an example, it can be any file name.

A skeleton makefile is given on the companion software, and a variant is included in the directory of each application. This will help you understand typical makefile usage, which is not difficult to learn.

The skeleton makefile reads as follows:

```
#**************************************************************
#   GNU MAKEFILE SKELETON
# This file can be used as a template for any project.
#**************************************************************

# DEFINES :
#**********

CFLAGS = -c -g -Wa,-alhds -O -fomit-frame-pointer -Wall -mlink -mfar
ASMFLAGS = -c -g -Wa,-alhds
LDFLAGS = -m  -I -mmu

APPLI = *The name of the main object file without extension*

Clist_SRC = *The list of the C source files separated by a space*
ASMlist_SRC = *The list of the assembler source files separated by a space*
Hlist_SRC = *The list of the .h include files in C files*
INClist_SRC = *The list of the .inc include files in assembler files*

# COMMON DEFINES :
#*****************
```

```
.SUFFIXES: # This command removes all previous suffixes
.SUFFIXES: .c .asm .ST9 .s .s9 .o .scr .u .br9 .dmp

HEX = $(APPLI).hex
EXE = $(APPLI).u
SCRIPT = $(APPLI).scr
list_OBJ = $(patsubst %.c,%.o,$(Clist_SRC)) \
           $(patsubst %.asm,%.o,$(ASMlist_SRC)) \
           $(patsubst %.ST9+,%.o,$(ASMlist_SRC))

$(HEX) : $(EXE)
   intel9 $< >$(HEX)

$(EXE) : $(list_OBJ) $(SCRIPT)
   $(LD) $(LDFLAGS) -T $(SCRIPT)

%.o:%.c $(Hlist_SRC)
   gcc9 $(CFLAGS) $< -o $@
%.o:%.asm $(INClist_SRC)
   gcc9 $(ASMFLAGS) $< -o $@
%.o:%.ST9+ $(INClist_SRC)
   gcc9 $(ASMFLAGS) $< -o $@
```

The syntax of this file looks complicated. In fact, you typically only have to change the parts of the text that are marked by two stars (*This text*) to refer to the actual file names of your project. More generally, you only need to change the .DEFINES section. The various items of the file are detailed below.

5.9.1.1 CFLAGS = -c -g -Wa,-alhds -O -fomit-frame-pointer -Wall -mlink -mfar

CFLAGS is a variable that contains the options that govern the working of the C-compiler. Referring to the option table of the compiler, it means:

Option	Meaning
-c	Output an object file for later linking.
-g	Include in the object file all necessary information for the debugger to allow it to use symbolic names.
-Wa,-alhds	Tells the gcc9 to transmit the option string -alhds to the assembler. These options makes the assemble generate the most complete assembly listing file to help finding problems more easily.

-O	Optimise the C compilation. This provides the most efficient code, both in terms of speed and memory size.
-fomit-frame-pointer	One of the parameters of optimization. Tells the compiler it can save a working register by not maintaining the stack frame pointer inside functions that do not need one.
-Wall	Output the warning messages that relate to most of the cases where the correctness of the source code is questionable, or may lead to inefficient code.
-mlink	tells the compiler to generate either link and unlink instructions in function prologue and epilogue, or use linku and unlinku instead if option -mparmusp is used.
-mfar	tells the compiler to generate a segmented application, meaning to generate calls and rets, both when calling a function and generating code for a function, except for static functions.

5.9.1.2 ASMFLAGS = -c -g -Wa,-alhds

ASMFLAGS is a variable that contains the options that govern the working of the assembler for assembly source files. These are the same options as that for the C-compiler, except for the optimization that does not apply in assembly language.

5.9.1.3 LDFLAGS = -m -I -mmu

LDFLAGS is a variable that contains the options that govern the working of the linker. The options are:

Option	Meaning
-m	Output a map file with the extension .map.
-I (uppercase i)	Generate in the code the table of the initial values of the variables to copy them into the RAM at start-up. This is detailed below in the start-up file paragraph.
-mmu	Tells the linker to generate an application for a ST9+ device; special relocations to handle new ST9+ instructions, new assembler operators and correct DPR translation are only possible under this option. Option -mmu also requests the linker to use 22-bit physical addresses when it generates a map listing, as well as to produce a table of DPR register assignments. By default, gcc9 invokes the ld9 linker with the -mmu option.

5.9.1.4 APPLI = *The name of the main object file without extension*

This variable must contain the name of the main object file and related files, these are:

File name	Type of file
<main file name>.u	Object file ready for loading into the emulator. This is defined in conjunction with the LD9 script file.
<main file name>.scr	The LD9 script file that governs the linker and defines the name of the output file.
<main file name>.map	The memory map file name.
<main file name>.hex	The hexadecimal file needed to program a PROM of an EPROM version of the ST9+.

Example:

```
APPLI = SMCB
```

generates SMCB.U, SMCB.MAP and SMCB.HEX using SMCB.SCR to define the linker behaviour.

5.9.1.5 Clist_SRC = *The list of the C source files separated by a space*

This variable must contain the complete list of the C source files involved in the program to build. Example:

```
Clist_SRC = main.c config.c serial.c calcul.c encoder.c
```

5.9.1.6 ASMlist_SRC = *The list of the assembler source files separated by a space*

This variable must contain the complete list of the assembler source files involved in the program to build. Example:

```
ASMlist_SRC = startup.asm interrup.asm
```

5.9.1.7 Hlist_SRC = *The list of the .h include files in C files*

This variable must contain the complete list of the .h include files used in all the C source files of the program to build. If a .h file itself includes another .h file, the latter must also be mentioned in the list, unless it is ensured that the lower level .h files will not be modified.

If a particular .h file is only used within a single .c source file, it is not wise to include it in this list, since each time it is changed, all the C source files will be recompiled. In this case, it may be useful to add a dependency rule for that file. See the explanation of the make rules in Section 5.9.1.9.

5.9.1.8 INClist_SRC = *The list of the .inc include files in assembler files*

This list is similar to the .h list above and the same remarks apply. It is intended for the include files used within the assembler source files.

5.9.1.9 Make Rules

A make rule is a statement that both tells which file is dependent on which other file, and the processing needed in order to update the result file when the source file has changed.

All files written under MS-DOS are marked in the directory with a date/time stamp. This allows the make utility to compare dates between files. If a file declared in a rule as being the result of a source file, and the source file has a later date than that of the result, the result must be regenerated. For example, in the following rule:

```
%.o:%.c $(Hlist_SRC)
    gcc9 $(CFLAGS) $< -o $@
```

The first line says that any file with extension .c is the source for the corresponding .o file, so that if the .c file is younger than the .o file, the source file must be recompiled. The .o file also depends on the .h files listed in the variable Hlist_SRC.

The second line says that the compilation is done using GCC9, with the options stored in the CFLAGS variable, that the input file ”$<” is the .c file (source file), and the output file ”$@”, specified by option -o, is the target file.

5.9.1.10 Using GCC9 to Generate Rules

The GCC9 has an option that allows scanning a source file and finding out all its dependencies. To do this you invoke it using the -MM option like this:

```
GCC9 -MM calcul.c
```

GCC9 will produce the following output on the screen:

```
calcul.o : calcul.c stepper.h ST90158.h stdio.h ctype.h
```

Actually, the file names are given with their complete path. It has been removed here for the sake of clarity. The list of include files shows only the first-level include files, not those included within them. You can also use the option -M that outputs the complete list, including files included in other files. Since most of the time these second level and deeper include files are supplied with the GNU9P tool chain, they are not likely to change. So it saves time when make is invoked by not scanning through all levels of inclusion.

Using this command for each file, the rules can be generated without having to look for the dependencies. This command is supplied with the Programmer’s File Editor configuration files delivered in the Companion software, and the result of the process appears in the output window. You can easily copy the text and paste it into the makefile if it is open at the same time.

5.9.2 Writing the Linker Command File using a Script File

The linker uses the linker command file[(8)](or Script File) to position the code and the variables correctly in the memory spaces. It defines the start and end of the ROM and RAM area(s), and gives the list of the object files to be linked. It also generates the appropriate labels to allow C variables to receive their initial values before the program starts.

Once the Script File is written the only thing you have to do is to add new file name or to change the mapping.

For example, in the example program given in the MMU section (Section 3.1.7), the script file reads as follows:

```
/* Define the output file format */
```

(8) ST9+ Family GNU Software tools release V2.12, Second Part: LD9.

```
OUTPUT_FORMAT(”a.out-st9”)
OUTPUT_ARCH(st9)
/* List of the object files (not the libraries) to be linked. */
/* The start-up file <crt9.o> must appear first. */
INPUT(crt9f.o wdtinit.o isrseg.o file0.o file1.o file2.o)
/* Define the memory configuration */
MEMORY {
/********************** Code segments ****************/
/* Define memory region ”text” = code segment 0. */
/* ”No” means no Data Page Register. */
   reset :   ORIGIN = 0x000000, LENGTH =  2K, MMU = NO
/* Define memory region ”segment0” = code segment 0 at address 0x0800 */
   segment0 : ORIGIN = 0x000800, LENGTH = 48K, MMU = NO NO NO
/* Define memory region ”segment1” = code segment 1 */
   segment1 : ORIGIN = 0x010000, LENGTH = 48K, MMU = NO NO NO
/* Define memory region ”segment2” = code segment 2 */
   segment2 : ORIGIN = 0x020000, LENGTH = 48K, MMU = NO NO NO
/*** Data pages ***/
/* Define memory region ”page11” = Data Page Register 1 */
   page11 :  ORIGIN = 0x044000, LENGTH = 16K, MMU = DPR1
/* Define memory region ”page12” = Data Page Register 2 */
   page12 :  ORIGIN = 0x048000, LENGTH = 16K, MMU = DPR2
/* Define memory region ”page13” = Data Page Register 3 */
   page13 :  ORIGIN = 0x04C000, LENGTH = 16K, MMU = DPR3
/* Define memory region ”ram” = Data Page Register 0 */
/* 0x20E000 = 0x20C000 + 0x2000 */
/*  page 0x20E/4=0x83 , 0x2000 for the start address */
   ram :    ORIGIN = 0x20E000, LENGTH = 512, MMU = DPR0
}
SECTIONS {
/* This line specifies the distance between the beginning and the end */
/* of the system stack. This gives the capacity of the stack in bytes.*/
/* The number is decimal. Like you can see the writing is as C language */
/* and means: If _stack_size is defined _stack_size equal _stack_size */
/* else _stack_size equal 256 */
```

```
   _stack_size = DEFINED(_stack_size)? _stack_size : 256;

/* .text contains start-up and image of initialized vars */
/* The section declared is <.text>. In the assembler program the call */
/* to this section is done with the write of <.text>. It’s mean that */
/* the code following this declaration will be put in the text section */
/* defined at address 0x0000 in the segment ”reset” which is 0.     */
   .text : {
/* The ”.” means the current address. So <_text_start=.> means */
/* _text_start=0 (reset segment start at address 0). */
      _text_start = .;
/* The two object file ”crt9f.o” and ”wdtinit.o” are place successively */
/* in this segment. */
      crt9f.o (.text);
       wdtinit.o (.text);
/* _etext =. equivalent to _text_start+length(crt9f.o+wdtinit.o)  */
      _etext =.;
/* Does start-up memory copy due to option -I. The Initial values */
/* is stored in this section. */
      DO_OPTION_I
/* _text_end equivalent to _etext+initial values */
      _text_end = .;
   } > reset
/* Map ”.text” section of ”file1.o” in the segment 1 and generate */
/* the file ”segment1.bk9” */
   segment1.bk9 : {
      _text_segmen1_start = .;
/* ”file1.o” the only file stored in the segment 1. */
      file1.o (.text);
      _text_segment1_end = .;
   } > segment1
/* Map ”.text” section of ”isrseg.o” and ”file2.o” in the segment 2 */
/* and generate the file ”segment2.bk9” */
   segment2.bk9 : {
      _text_segment2_start = .;
/* ”isrseg.o” and ”file2.o” two files stored in the segment 2 */
      isrseg.o (.text);
      file2.o (.text);
      _text_segment2_end = .;
   } > segment2
/* Map ”.text” section of ”file0.o” at address 0x0800 in the segment 0 */
/* and generate the file ”segment0.bk9”.*/
   segment0.bk9 : {
      _text_segment0_start = .;
/* ”file0.o” the only file stored in the segment 0. */
      file0.o (.text);
      * (.text)
      _text_segment0_end = .;
   } > segment0
/* Map ”.bss” section in the page ”ram” addressed by DPR0  and start */
/* at address _bss_start=_data_end. */
/* .bss contains all uninitialized variables and system stack. */
   .bss : {
      _bss_start = .;
/* Uninitialised data of the 5 files stored in the .bss section */
       crt9f.o (.bss COMMON);
       wdtinit.o (.bss COMMON);
      file0.o (.bss);
      file1.o (.bss COMMON);
      file2.o (.bss COMMON);
      _bss_end = .;
/* System stack placed just after the uninitialized data */
      _stack_start = DEFINED(_stack_start) ? _stack_start : .;
      _stack_end = _stack_start + _stack_size;
/* User stack placed just after the system stack */
      _user_stack_start = .;
      _user_stack_end = .;
   } > ram
/* Page 11, 12 and 13 contain constants, ROM memory. */
/* The first data address is 0x4000 and point on 0x044000 using*/
/* the DPR1 register. */
    page11.bk9 : {
     _data_segment0_start = .;
      crt9f.o (.data);
      wdtinit.o (.data);
     file0.o (.data);
     _data_segment0_end = .;
   } > page11
/* The first data address is 0x8000 and point on 0x048000 using */
/* the DPR2 register. */
   page12.bk9 : {non initial
      _data_segment1_start = .;
     file1.o (.data);
      _data_segment1_end = .;
   } > page12
/* The first data address is 0xC000 and point on 0x04C000 using */
/*  the DPR3 register. */
   page13.bk9 : {
     _data_segment2_start = .;
     file2.o (.data);
      _data_segment2_end = .;
   } > page13
/* Map ".data" section in the page "ram" addressed by DPR0 */
/* and place the initializer data in section ".data" of the file */
/* "segment.u".*/
   .data : {
     _data_start = .;
     *(.data)
     _data_end = .;
   } > ram
/* That's all folks */
}
```

For more details on the Command File see the GNU Software Tools User Manual at the Command Language chapter.

5.9.3 Writing the Start-Up File

This file is the very first one executed at power on. It does the necessary initialisation to allow the processor to start. It differs according to whether the program includes modules written in C or not.

In all cases, it includes two main parts:

– The reset, exception and interrupt vectors.

– The initialisation code that is mandatory to allow the program to start.

5.9.3.1 Vector Table

The vectors consist of the Reset vector, the Divide by zero vector and the other interrupt vectors. They are placed at defined addresses:

– The Reset vector must be located in the first word of the first segment.

– The Divide by zero trap must be placed at the address 2 in each segment where a program uses the division. Each segment containing programs using division must have its own Address trap branches to a local routine making only a far call ”CALLS DIVIDE_BY_ZERO” to the Divide by zero service routine located in only one segment.

– All the other interrupt addresses must be placed in the Interrupt Segment. In ”ST9+” mode, Interrupt Service Routines can make far calls to other segments.

They are organised as follows:

Segment 0:

Address	Cause	Point to
0	Reset (action on the reset pin, Watchdog reset or Software reset).	The start of the initialisation code. The reset address is the start-up address.
2	Divide by zero	Address of the routine in segment 0 that handles the case where a division by zero has occurred.

Segment n (not Interrupt Segment):

Address	Cause	Point to
2	Divide by zero	Address of the routine in segment n that handles the case where a division by zero has occurred.

Interrupt Segment:

Address	Cause	Point to
2	Divide by zero	Address of the routine in segment ISR that handles the case where a division by zero has occurred.
4	Top level interrupt	The interrupt service routine for the top level interrupt.
...	Interrupt	Interrupt Service Routine addresses.

The vectors above are placed at these addresses by hardware. You can place the following ones at will, except that for each peripheral they must obey some rules like being a multiple of a given number, e.g. 8 for the SCI.

Address	Cause	Point to
n	First cause for the peripheral whose IVR is set to n	The routine that handles the first interrupt cause for that peripheral.
n+2	Second cause for the peripheral whose IVR is set to n	The routine that handles the second interrupt cause for that peripheral.
etc.		

To force the linker to effectively position these vectors from address zero, the start-up module must be in first place in the module list in the linker command file.

5.9.3.2 Initialisation Code

The initialisation code mainly has to initialise the core. The core has a few control registers that must be set to correct values or the program will fail to start. They are listed in the table below. They are in the order they are initialised in a typical start-up file, though this order is somewhat arbitrary.

Register number	Register name	Function(*)	Comments
235	MODER	Selects the space where the stacks reside (register file or memory), main clock divider on/off, clock prescaler division rate.	If power and electromagnetic interference are not a concern, the prescaler can be set to zero so that the processor runs at full speed. Otherwise, the speed/ consumption trade-off can be handled at will according to the needs of the program.
231	FLAGR	Selects the single space of twin-space mode for the memory.	In the ”ST9old” mode the DP bit of this register selects program memory when cleared, or data memory when set. It must be changed using the sdm and spm instructions. Typically, one of these is used at the beginning of the program and remains unchanged afterwards. On some occasions, it may be changed, for example to access tables of constants in ROM if two spaces are used. Therefore in ”ST9+” mode, DP must be set with the sdm instruction (used only once in the start-up program) to set the use of DPR register to address data memory. The spm instruction must not be used. To access data through the CSR register, use the ldpd, lddp or ldpp instruction.

Register number	Register name	Function(*)	Comments
238-239	SSP	Position of the top of the system stack in memory or register file	No subroutine or function call may be performed before the initialisation of SSP and MODER. Note that since a call first decrements the stack pointer by two and then writes the return address, if the stack is positioned in the register file, it may be initialised to the register number of the top of stack + 1 to save stack space.
236-237	USP	Position of the top of the user stack in memory or register file. Unused if only one stack is used.	
232-233	RP0-RP1	Selects the mode for the working registers as well as the absolute register numbers for r0 and r8.	These registers are set using the instructions srp, srp0 and srp1. See the description of working registers in Section 3.1.4.
230	CICR	Enables the MFTs, enables the interrupts, selects the arbitration mode (concurrent mode or nested mode), and the priority level of the main program.	The GCEN bit of this register enables all the MFTs at once. It may also be set once the MFTs are all initialised, if it is desired that they be synchronised or that they do not start before the remainder of the program is ready. The IEN bit enables all the interrupts at once. Again, it may be set now, or only when the remainder of the program is ready to process them. The IAM bit selects either Concurrent Mode or Nested Mode. It is better to do it now. The three bits CPL2 to CPL0 set the level of the main program. Its value depends on the structure of the program. Typically, it is set to 7, since the main program is likely to be the low-priority process. See the paragraph on interrupts in Section 3.4.
234	PPR	Sets the page number for the paged register group; set to zero.	This register will probably be changed several times during the execution of the program. It is first set now, if the WCR needs to be initialised.
250 page 0	WDTPR	Prescaler register of the WatchDog Timer.	If the WDT is used as a watchdog(†), it should be initialised before the WDGEN bit is cleared in the WCR register below. In this case, it is advisable to initialise it right now. See note for values.
248-249 page 0	WDTHR+WDTLR	Reload register of the WDT.	Same note as above.

Register number	Register name	Function[*]	Comments
251 page 0	WDTCR	Mode control register of the WDT.	Same note as above.
252 page 0	WCR	Starts (or not) the watchdog function of the WDT, selects the wait states for external memory access independently for upper and lower memories.	If you use the Watchdog function, first initialise the Watchdog Timer. You can start it later using the WDGEN bit of this register, but no protection is on before this time. The external memory access is set by default to the maximum number of wait states to allow the program to start. You should reduce it to the exact number required by the type of memory to achieve maximum performance.

(*) Some functions of a register may not be mentioned if not relevant for the initialisation.

(†) ST90158 - ST90135 Data Sheet; §9

Once you have set these registers, the core is in good shape to start work. The next task is to initialise the data memory and/or register file by entering a loop that writes zeroes into all locations used for data storage. This may seem superfluous, but there are two reasons for this:

– In C language, any non-initialised variable is supposed to contain zero at as an initial value

– In any case, this makes the program behaviour reproducible, even if not all variables are initialised explicitly.

Then, if you are using C language and link the program with option -I, the table of initial values for the initialised variable (in the C language sense) is copied to the location in RAM where the variables are positioned. The linker puts the initial values in the same order as the variables in RAM, so a mere block copy is sufficient for this initialisation.

Eventually, the entry point of the main program written in C or assembler can be called. The main program should be a closed loop and should not return. Typically in the start-up code, the call to the main routine is followed by a halt instruction.

5.9.3.3 Start-up C source file

C start-up code for segmented applications are delivered; in fact, 4 start-up files for small code models, 4 start-up files for large code models and 4 start-up files for ST9 compatibility models are provided. A complete source structure is also provided that helps customize and regenerate these 12 start-up files from a single file (crt9.c). Regeneration can be performed from any application directory using the vpath mechanism of gmake. (see the ST9+ GNU C TOOLSET RELEASE 4.2, C Start-up Files chapter for more details).

5.9.3.4 Start-Up File Example

This example is taken from MMU application described in Section 3.1.7. It can be easily modified for other purposes, since the overall structure remains the same. The parts that are likely to change are:

Vector table

Core mode

Location of the stack(s)

```
; Template start-up file for ST9+ microcontroller applications
;  - compiled with -mfar option for large code model
;  - linked with  option -I to copy initial .data values from .text section
;
; Note that an instruction which loops forever is generated after ’main’.
;
; File crt9f.asm  has been automatically generated from ’crt9.c’
; You may decide either to customize this particular version or to modify
; the generic file ’crt9.c’ and regenerate all start-up files with
;’gmake’.
;
; Refer to instructions in the ”ST9+ GNU Toolchain Release Note 4.2”.
; See also macro-definitions in ’options.h’ and explanations in
;’readme.txt’.
;
.include ”page_0.inc”
.include ”system.inc”
.include ”mmu.inc”
;
;  CICR = IT disabled + Nested Mode + CPL = 7
;  MODER = both stacks in memory + clock divided by 2
;  WCR = zero wait state + watchdog disabled
;
   INIT_CICR = 8fh
   INIT_MODER = 20h
   INIT_WCR = 40h
   .text
   .word    __Reset
   .blkb    50          ; reserve room for the interrupt vectors
;
;  Reset routine
;  WARNING : it is important to set rr12 to a NON zero value.
;  This is because for the debugger, a null frame pointer means
;  that the function has no parent frame (ie cannot go up). This is OK
;  for the main routine, but not for routines called by main.
;  If -fomit-frame-pointer is used for compiling, rr12 could not
;  be set in main, nor in functions called by main...
;  Furthermore, it seems a good idea to initialize the current frame
;  pointer to the current stack pointer.
;
   .global __Reset
   .global __Halt
__Reset:
;
;  Initialisation of registers controlling external memory interface
;  EMR1 reset value is x000-000M
;  MC, DS2EN, ASAF, NMB, ET0, BSZ should be checked against user
;  memory configuration and EMR1 set accordingly
;
;  EMR2 reset value is M000-1111
;  ENCSR is 0, which selects ST9 compatibility mode for interrupt
;  handling.
;  DMEMSEL, PAS1, PAS0, DAS1 and DAS0 should be checked against user memory
;  configuration and set complementarily to WCR in page #0 (see below)
;  DPRREM is forced to one to have DPRi registers accessible in group E
;
           spp         #MMU_PG
           or          EMR2, #EMR2_dprrem ; remap data page registers
;
;  Initialisation of registers controlling the Reset and Control Clock
;  Unit
;  Check initial values of registers PCONF, SDCTL, SDRATH
```

```
;
          ; spp       #55            ; uncomment if necessary
;
; Page 0 register initialisation
;  Check initial values of SPICR, SPIDR, WCR, WDTPR, WDTLR, WDTLH,
;  NICR, EIVR, EILPR, EIMR, EIPR and EECR
;
          spp         #0
          ld          WCR, #INIT_WCR ; WCR = zero wait state
;
;  System registers initialization in group 0xE (R224 to R239)
;  init clock mode and select external stacks in data memory
;  Set register pointer to group 0xD
;
          ld          MODER, #INIT_MODER
          ld          CICR, #INIT_CICR
          srp         #26
;
;  No more inclusion of specific pdselxx.asm files
;
;
;  Initialization of DPR registers
;
          ld          DPR0, #0x83
          ld          DPR1, #0x11
          ld          DPR2, #0x12
          ld          DPR3, #0x13
;
;  Stack registers may be initialized after DPRs have got their initial
   values
;  DPR registers pointing the stack pages should better remain fixed
;
          sdm                        ; set data memory
          ldw         SSPR,#_stack_end ; setup stack
          ldw         USPR,#_user_stack_end ; setup user stack
```

```
;
;  Section to initialize the data, bss and stack sections.
;  Note: ld9 creates the following symbols :
;
;  _data_start  points to the start of the run time data area,
;  _data_end    points to the end of the run time data area,
;  _bss_start   points to the start of the run time bss area,
;  _bss_end     points to the end of the run time bss area,
;  _text_start  points to the start of the text segment,
;  _text_end    points to the end of the text segment,
;  _stack_start points to the start of the stack segment,
;  _stack_end   points to the end of the stack segment.
;
__InitDataBss :
;
;  Note: option I must be used with ld9 in order to get the
;  initialized data information at the end of the text section (in ROM).
;
;  In that case, '_text_end - (_data_end - _data_start)' gives the start
;  address of the initialized data information in the text segment.
;
;    Init .data area
;
          ldw       rr0,#_data_start ; start of run-time data area
          ldw       rr4,#_data_end ; end of run time data area
          subw      rr4,rr0          ; rr4 = length of initialized data
          jrz       no_data          ; if empty
          ldw       rr2,#_text_end ; end ROMed data area
          subw      rr2,rr4          ; start of ROMed data area
init_data:
          lddp      (rr0)+,(rr2)+ ; init data section
          dwjnz     rr4,init_data
no_data:
;
;     Init .bss section
;    (here r4 = 0)
;
          ldw       rr0,#_bss_start ; start of run-time bss area
          ldw       rr2,#_bss_end ; end of run time bss area
          subw      rr2,rr0         ; rr2 = length of bss area
          jrz       no_bss          ; if bss is empty
init_bss:
          ld        (rr0)+,r4  ; clear whole bss section (Data space)
          dwjnz     rr2,init_bss
no_bss:
;
;    Init stack section (not really necessary, but cleaner)
;    (here r4 = 0)
;
          ldw       rr0,#_stack_start ; start of run-time stack area
          ldw       rr2,#_stack_end ; end of run time stack area
          subw      rr2,rr0         ; rr2 = length of stack area
          jrz       endinit         ; if stack is empty
init_stack:
          ld        (rr0)+,r4  ; clear whole stack section (Data space)
          dwjnz     rr2,init_stack

endinit:
;
;    Enable interrupt and call main routine
;
          ei                        ; enable interrupts
          ldw       rr12,SSPR  ; make sure rr12 is NOT zero
          calls     main
__Halt:
          jr        .              ; halt or jp .
;
;    End of crt9f.asm
```

Note: The start-up file initialises the ”ST9old” mode (ENCSR bit of EMR2 to 0) to be compatible with the old ST9 version. In the main program you have to initialise this bit to 1 for ”ST9+” mode if you want the MMU working with segments during interrupts.

5.9.4 Constants and Initialised Data

This is a point that deserves some care, especially if your program includes a large amount of constants.

Here is the point. In assembler, variable data are located in RAM and constant data are put in ROM. You can do this easily using the .text, .data and .bss directives.

By convention, the .data segment is dedicated to initialised variables in C language. So it is advisable to use a .bss section for the variables defined in an assembly module. Then, accessing variable data and constant data is done the same way in assembler.

The C language has been designed for use on computers. In these machines, the program resides in RAM and the concept of ROM does not exist. Constants and initialised variables are defined in the program file and merely copied in RAM at program loading.

When using C in an embedded application, it is necessary to take into account the difference between ROM and RAM. ROM, as its name implies, is read-only, and RAM has an undefined content at power on. So, C compilers designed to produce ROMable code must turn this problem around. The solution varies from one compiler to another, since the ANSI standard has not specified anything about embedded application programs.

In the GNU9P compiler, the following technique is used:

Non-initialised variables are mapped straight to RAM in the .bss section. This raises no question.

Initialised variables are also mapped to RAM in the .data section, but their initial value is mapped to ROM. Thus all the .data section is the storage area for these variables. The compiler and the linker take care to lay the initial values out in the .text section in the same order as they have in the .data section, so that a simple memory to memory copy routine can initialise the .data section at the beginning of the program. See Section 5.9.3.4 on the start-up module.

The following example, taken from the linker command file, shows how the initial values are handled. The command DO_OPTION_I tells the linker to include the values at the end of the .text section, after the code itself, and between the labels _etext and _text_end. These labels are public, and they are used in the initialisation code to do the copying.

The relevant part of the linker command file looks like this:

```
.text : {
        _text_start = .;
```

```
           *(.text)
           _etext = .;
           DO_OPTION_I
           _text_end = .;
   } > rom
```

And the copy routine in the start-up code:

```
; Init data area
;
    ldw    rr0,#_data_start      ; start of run-time data area
    ldw    rr4,#_data_end        ; end of run time data area
    subw   rr4,rr0               ; rr4 = length of initialize data
    jrz    no_data               ; if empty
    ldw    rr2,#_text_end        ; end ROMed data area
    subw   rr2,rr4               ; start of ROMed data area
init_data:
    lddp   (rr0)+,(rr2)+         ; init data section
    dwjnz  rr4,init_data         ;
no_data:
```

The lddp instruction allows you to transfer data easily from program memory to data memory. Without this instruction, the transfer of one byte would have needed to write something like:

```
   ld  r4,(rr2)+; get initial value
   ld  (rr0)+,r4; write into storage area
```

Note 1: To include the initial values at the end of the .text section, the linker must be invoked with the -I option.

This method works fine, and suits most applications. The only case where it could be a source of trouble, is when large amounts of constant data are used. With the process described above, every byte of initialised data takes two bytes in memory: one in RAM and one in ROM. In the case of constant data, i.e. data that are only read and never written, this can lead to a waste of valuable RAM. This problem can be alleviated in several ways that are described in the documentation(9).

(9) ST9+ Family GNU C Compiler; §5.6

5.10 TESTING APPLICATIONS USING THE EMULATOR

The debugger program runs on a host PC under Windows and allows you to watch program execution step by step, to examine the value of variables and registers, and to place break-points.

To use the emulator, you have to first configure the hardware and software.

Refer to the User Manual of your emulator for information on configuring the hardware.

The software configuration is done by two configuration files. The first one is HARD-WARE.GDB. A file with this name is supplied in the GNU9P\WGDB9XXX directory, it is an ex-ample file capable of supporting several different ST9+ family members. This file must be copied into the directory of the application and customised to fit its requirements.

Other commands may be added in this file. Please refer to the Windows Debugger for ST9+XXX Emulator User Manual.

The HARDWARE.GDB file is read before the application is loaded. You must define the memory mapping in this file, to avoid errors such as the program loading in an address range defined as non-existent by default.

The second file, <appli>.GDB, (where <appli> is the name of the executable .U file), has the same function and is written in the same language. This allows you to execute more com-mands once the program is loaded in memory. This is necessary in cases where MMU works with more than one segment, since the program is shared between several object files, with the .BK9 extension. It has to be loaded in several passes. To do this the LD9 linker produces another file with the extension .LD9 along with the .BK9 files. This file is a batch command that directs the emulator to load the right file in the right place. This .LD9 file must be invoked in the <appli>.GDB file.

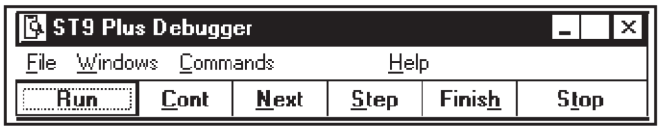
5.10.0.1 Running the Debugger

As an example, we shall run the short program described in the Chapter 3 as ”Application of the Watchdog/Timer as a Pulse Width Modulation generator”. This program has already been used to illustrate the assembler. To watch the correct working of the program, it is only neces-sary to wire an LED in series with a resistor between P7.6 and Vcc (anode towards Vcc). We suggested you reproduce the actions described to see how things work.

Once the Emulator is connected, configured and powered on, the debugger program can be started by double-clicking on its icon. The welcome panel then appears, indicating the soft-ware version and the message:

”Please wait while trying to connect with the emulator”.

The main control panel appears then. A convenient place to put it is in the upper lefthand corner of the screen. It looks like this:



The menus give the following facilities:

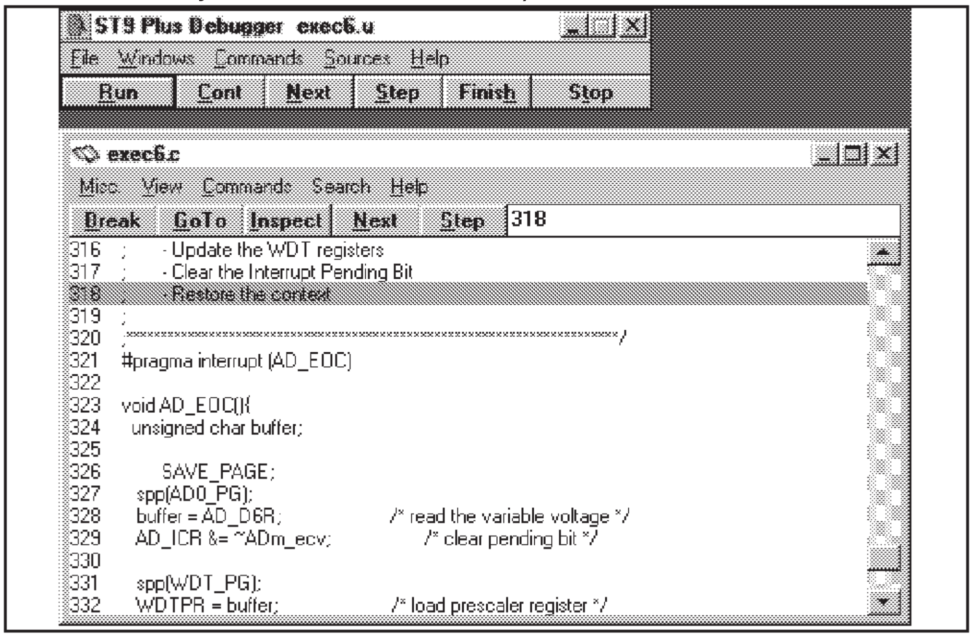
File	This menu allows you (as in any Windows application) to select the file to use, or to re-call one of the last files used. A Preference options allows you to modify the screen's appearance.
Windows	This menu allows you to open as many windows as you wish, to display various things like source files, the disassembled code, the memory contents, the register values and also the variables with their names and value in so-called "inspect windows".
Commands	This menu allows you to reset the program, or the program and the emulator as a whole, to show in the source window the instruction where the PC is pointing (i.e. the next in-struction to execute), and more.
Sources	This menu presents the list of the source files that make up the program. Choosing one of these opens a new window to view the file.

The buttons are:

Run	Resets the chip and runs the program from the beginning.
Cont	Continue an interrupted program. (after Break, Stop, Step).
Next	Execute a single instruction and execute function or subroutine calls at once.
Step	Execute a single instruction. If it is a function or subroutine call, execute the first statement or instruction of it.
Finish	Finish the current subroutine and stop after the return instruction.
Stop	Stop the currently running program. Use Cont to resume.

Using the File/Open menu command, select the executable file to debug. In this guide, these files have the extension ".u". Here we shall use the executable file WDTRECT.U.

When this file is selected, a window appears saying briefly "Downloading .text section of file WDTRECT.U". Then a window opens showing the source file containing the first instruction to execute. Here there is only one file, which is WDTRECT.ASM. The screen now looks like this (the windows can actually be set and sized at will):


ST9 Plus Debugger exec6.u
File Windows Commands Sources Help
Run Cont Next Step Finish Stop
exec6.c
Misc View Commands Search Help
Break GoTo Inspect Next Step
318

```
316  ;      - Update the WDT registers
317  ;      - Clear the Interrupt Pending Bit
318  ;      - Restore the context
319  ;
320  ;*********************************************************************/
321  #pragma interrupt (AD_EOC)
322
323  void AD_EOC(){
324    unsigned char buffer;
325
326         SAVE_PAGE;
327     spp(AD0_PG);
328     buffer = AD_D6R;               /* read the variable voltage */
329     AD_ICR &= ~ADm_ecv;                /* clear pending bit */
330
331     spp(WDT_PG);
332     WDTPR = buffer;                /* load prescaler register */
```

This window shows the source text of the program. One line at a time may have zero to three of the following attributes: selected, current, and breakpoint.

Selected	The line currently selected is shown in colour. To select another line, just click once on it: the colour highlighting will move to it.
Current	The current line is the next one to be executed. It is underlined, as is line 85 in the figure above.
Breakpoint	The line on which the breakpoint has been placed is shown in bold. No breakpoint is currently set.

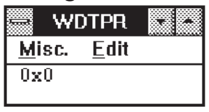
The top of the Source window has the following buttons:

Break	Sets or removes a breakpoint from the currently selected line of source.
Go To	Starts execution from the current instruction until the selected line is reached.
Inspect	Gives information on the identifier on which the caret currently stands.
Next	Executes an instruction or statement, without descending into functions or subroutines.
Step	Executes one instruction and descends if needed into the function or subroutine called.

The Next and Step buttons are the same as that of the main control panel as described above.

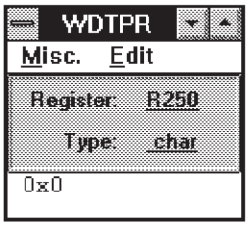
The inspect command is very useful. To use it, move the mouse cursor to an identifier representing some data (variable name, register name), and click once anywhere within the name of that identifier, so that the caret moves to it. For example, locate line 29 and put the caret in the word WDTPR. The line will look as follows:

Then press the Inspect button. The following window appears:

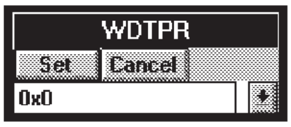
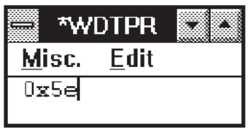


0x0

The Misc. menu gives the following options:

Hexadecimal	If checked, the value is displayed in hexadecimal. If not, the value is shown in decima if it is a simple variable or as a character string if it is an array of bytes.
Hot	The value shown is that at the time of the creation of the window. If the window is made Hot (the Hot word is checked and the background of the window is yellow), the display of the value will be refreshed after each stop in program execution.
Info	If checked, the window displays added information on the location of the variable as fo-lows:  Register: R250 Type: char 0x0
Close	Closes the window (same as double-clicking on the control button).

The Edit menu gives the following options:

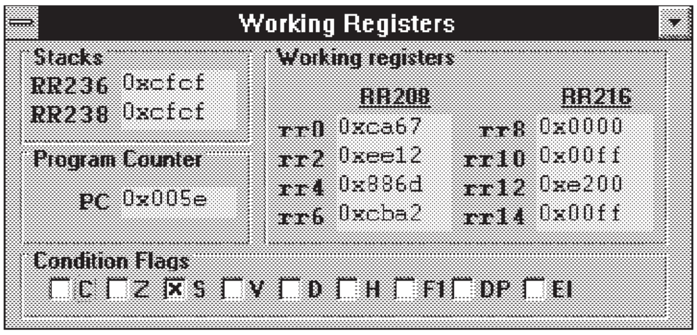
Copy	Copies the selected text to the clipboard to be pasted later (standard Windows copy func-tion).
Modify	Opens the following window:  0x0 and allows you to modify the value of the variable. Pressing Set changes the value. Press-ing Cancel does nothing. The change window is closed in both cases. Note: You can open this window just by trying to modify the contents of the Inspect window.
Select all	Selects the contents of the window for copying.
Dump	Does the same action as the Windows/Dump command of the main control window.
Inspect	Uses the value of the identifier as a pointer in data memory and opens another window to show the pointed data:  0x5e In this case, address 0 in memory contained 5Eh.

The Inspect windows show the value of a simple variable in decimal or hexadecimal and, if the variable is a character string, the text of that string. It can also display C-language structures with the member names and their contents.

Now we are familiar with the main option, let’s do a few exercises using the program we have already loaded.

First, since we can see only the last source lines at line 29 on the display, we do not know where the program starts. To show the next (here the first) instruction to execute, we use the option Commands/Display PC of the control window. Line 85 is now displayed at the centre of the source window.

We will execute a few instructions in step mode, and watch what happens to the core of the microcontroller. To do this, we add a new window using the option Windows/Registers/ Working registers in the control window. The following window shows then:



RR236 0xcfcf
RR238 0xcfcf

PC 0x005e

	RR208		RR216
rr0	0xca67	rr8	0x0000
rr2	0xee12	rr10	0x00ff
rr4	0x886d	rr12	0xe200
rr6	0xcba2	rr14	0x00ff

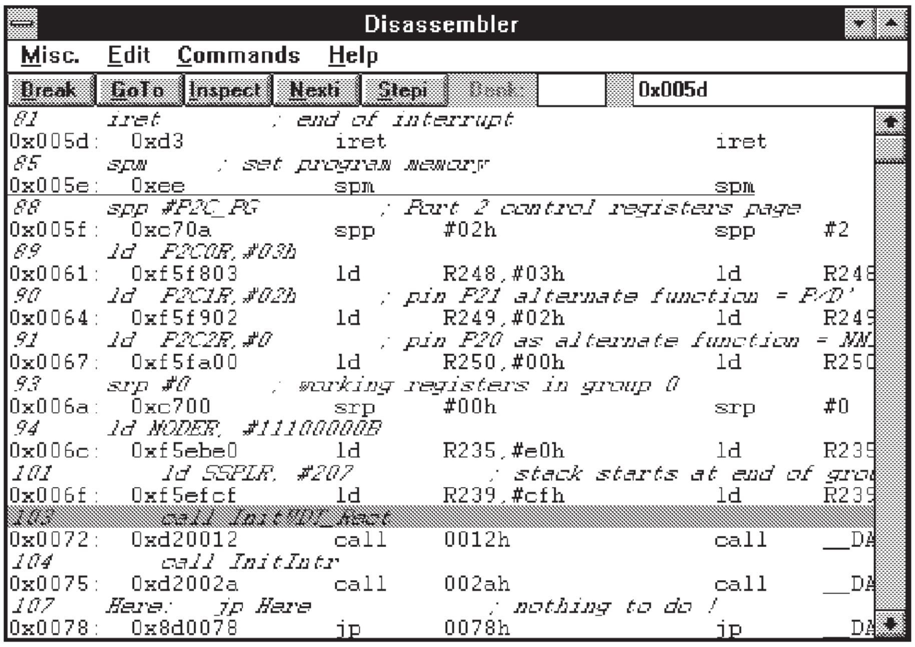
☐ C ☐ Z ☒ S ☐ V ☐ D ☐ H ☐ FI ☐ DP ☐ EI

Press the Step button.

In the Source window we can see on that the Current and Selected attributes have moved to line 88. The only change in the core is that of the PC that is now displayed in bold, with its new value 5Fh.

If we want to execute the previous instruction again or skip to the current one, we can change the value of the PC by merely typing a new value over the current PC value in the Working Registers window. For example, let us correct the value 5f into 5e and press Enter. The Current and Selected line is now line 85 as before.

Let us execute now until line 103 not included. We'll scroll the Source window and select line 103 by clicking it once. Then we press the Go To button. The program stops and a new window is open, the Disassembler window, which looks like this:



```
81        iret          ; end of interrupt
0x005d:   0xd3              iret                                  iret
85        spm       ; set program memory
0x005e:   0xee              spm                                   spm
88        spp #P2C_PG               ; Port 2 control registers page
0x005f:   0xc70a            spp      #02h                         spp      #2
89        ld  P2C0R,#03h
0x0061:   0xf5f803          ld       R248,#03h                    ld       R248
90        ld  P2C1R,#02h            ; pin P21 alternate function = P/D'
0x0064:   0xf5f902          ld       R249,#02h                    ld       R249
91        ld  P2C2R,#0              ; pin P20 as alternate function = NM
0x0067:   0xf5fa00          ld       R250,#00h                    ld       R250
93        srp #0        ; working registers in group 0
0x006a:   0xc700            srp      #00h                         srp      #0
94        ld MODER, #11100000B
0x006c:   0xf5ebe0          ld       R235,#e0h                    ld       R235
101           ld SSPLR, #207            ; stack starts at end of gro
0x006f:   0xf5efcf          ld       R239,#cfh                    ld       R239
103           call InitWDT_Rect
0x0072:   0xd20012          call     0012h                        call     __DA
104           call InitIntr
0x0075:   0xd2002a          call     002ah                        call     __DA
107       Here:    jp Here              ; nothing to do !
0x0078:   0x8d0078          jp       0078h                        jp       __DA
```

The Go To command actually sets a temporary breakpoint at the selected line. The response to a breakpoint stop is the opening of the Disassembler window. We see that the call instruction at line 103 has not been executed.

We can now go down into that subroutine by pressing Step. The Current and Selected line has moved to the first instruction of the subroutine in both the Source window, and the Dissassembler window. We can choose either window to follow the program's progress. Right now, the contents of both windows is not very different because this program is written in assembler; but if it were written in C-language, the effect of the Step and Next buttons would be different. In the Disassembler window, stepping involves only one processor instruction. In the Source window, stepping would involve one full C statement.

Let us execute until after the subroutine return. We are now back at line 104, which is again a subroutine call. Let's press Next. The Current line moves to line 107, and we did not see the instructions within the subroutine.

We shall now use the breakpoints. Let's locate the ReloadWDT label. We can either peek in the source text or use the option Search/Find... in the Source window. For example let's put a breakpoint on line 66. To do this we select this line and press the Break button. This line appears in bold.

We can now start the execution by pressing the Cont button of the control window. When the execution stops, the Current line has moved to the line where the breakpoint has been set. If we press Step now, we can see that r0 has changed (it is shown in bold) and it has a new value that comes from the WDTCR.

We can now remove the breakpoint by selecting the line where it was set and by pressing the Break button. The line is now shown normally. Now let’s press the Cont button. The program starts for an infinite duration, and we can watch the LED flashing.

By pressing the Stop button we can stop the program and see where it has been stopped.

6 DEVELOPMENT TOOL REFERENCE INFORMATION

6.1 TABLE OF TOOL-RELATED FILES

This table lists the most important files for each main tool. It gives the directory path and a description of the purpose of each file. You can install the tools in a directory of your own choosing. Subdirectories with predefined names will be created below this main directory. Under DOS or Windows 3.11, you can use the DOS command SUBST to give the installation directory the alias D: which is used by convention.

6.1.1 DOS tools

These tools run under DOS or in a DOS box under Windows. As a shortcut you can call them from the programming environment.

The GCC9 entry point is actually a dispatcher program that calls the appropriate executable files. A summary of the various command line options is given in the next paragraph.

6.1.1.1 C compiler

This tool translates C-language source files into object files.

Main file	GCC9.EXE
Secondary files	CPP9.EXE CC9.EXE TR9.EXE GAS9.EXE
Invocation	GCC9 -c <options> <filename>
Path	C:\GNU9P\BIN
Other files used	Header files *.h
Their path	C:\GNU9P\INCLUDE and C:\GNU9P\INCLUDE.ST9+

6.1.1.2 Assembler

This tool translates assembly language source files into object files.

Main file	GCC9.EXE
Invocation	GCC9 -c <options> <filename>
Secondary files	TR9.EXE GAS9.EXE
Path	C:\GNU9P\BIN

Other files used	Header files *.inc
Their path	C:\GNU9P\INCLUDE and C:\GNU9P\INCLUDE.ST9

6.1.1.3 Linker

This tool bundles the various object files together to produce an executable and debuggable file.

Main file	LD9.EXE
Path	C:\GNU9P\BIN
Invocation	LD9 <options> <filename>
Secondary files	if the invocation option defines it, the link script file <filename>.SCR
Other files used	Library files *.l
Their path	C:\GNU9P\BIN

6.1.1.4 Make Utility

This tool does the project housekeeping. It checks for the date of the source files, recompiles those that have changed and re-links the whole project if needed.

Main file	GMAKE.EXE
Path	C:\GNU9P\BIN
Invocation	GMAKE
File to work with	MAKEFILE ...
Path	User’s application directory
Secondary files	None

6.1.2 Windows Tools

These tools run under Windows only. They are usually installed as an icon in Program Manager.

You can call them from the programming environment for convenience.

The GCC9 entry point is actually a dispatcher program that calls the appropriate executable files. A summary of the various command line options is given in the next paragraph.

6.1.2.1 Emulator User Interface

Main file	WGDB9XXX.EXE
Invocation	WGDB9XXX. Usually done by an icon in Program Manager
File to load from file menu	<Filename>.U in the user's application directory
Secondary files	Many .EXE, .DLL, .HLP etc. files
Path	C:\GNU9P\ WGDB9XXX
Other files used	WGDB9WIN.INI in the user's application directory WGDB9WIN.INI in default Windows directory C:\GNU9P\ WGDB9XXX\GDB9XXX.INI HARDWARE.GDB in the user's application directory

6.2 GNU TOOLS AND OPTIONS

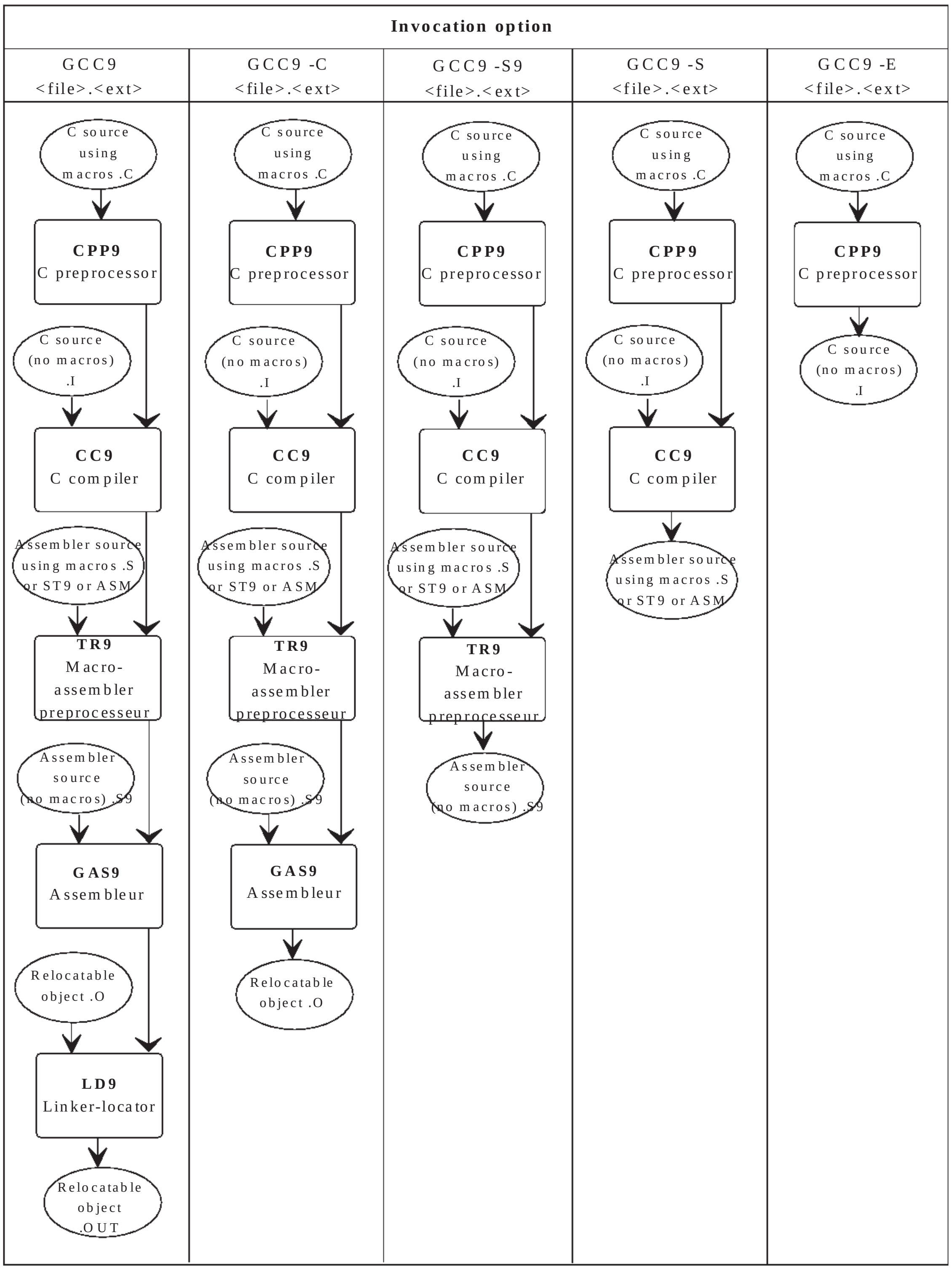
The GNU tools are made of five processing blocks that process the input file successively. Each step uses one file type on input, and produces another file type on output. Each file type has its own extension.

These steps are:

Input file	Ext.	Processing block	Type of processing	Output file	Ext.
A C source with macros used.	.C	CPP9	The # pseudo-statements are expanded. Include files are made part of the final file.	C source with macros fully expanded.	.I
A C source with macros fully expanded.	.I	CC9	The C statements are translated into assembler statements.	Assembler file with macros used.	.S
An assembler file with macros used.	.S or .ASM	TR9	The macros and equate statements are expanded. Included files are made part of the final file.	Assembler file with macros fully expanded.	.S9
An assembler file with macros fully expanded.	.S9	GAS9	The assembler statements are translated into machine language, but the addresses are floating.	Relocatable object.	.O
A list of relocatable object files.	.O	LD9	All the machine language files are linked, and the floating addresses are fixed.	Located object.	.OUT

This looks complex, but in practice it is not, because all the tools are grouped into the main control tool named GCC9 that starts these tools successively, as required. You can select the part of the processing you wish to use. You select the tools in the GCC9 invocation line using the input file extension and option switch. The following table shows the 15 allowed combinations of input file type and invocation options.

Figure 46. GCC9 Allowable File Types and Options



6.3 MEMORY ORGANISATION: COMPILE AND LINK OPTIONS

There are three main memory allocation options:

All data in registers, one or two stacks

Some data in registers, some in memory, one stack in memory (user stack not used)

Some data in registers, some data in memory. Two stacks: system stack either in registers or in memory, user stack in memory. This is a double option.

This choice implies changes in the following places:

– Stack initialisation code

– Compile directives

– Link script

– Libraries used at link time

– STARTUP code

6.3.1 Stack Initialisation Code

You must specify the stacks as residing either in the register file or in the data memory. Three registers are involved:

MODER Register

Bits USP and SSP indicate for the user stack and the system stack whether they are in the register file (if ”1”), or in memory (if ”0”):

SSP	**USP**	DIV2	PRS2	PRS1	PRS0	BRQEN	

SSP Register Pair

This is where you set the initial address of the system stack. If the stack resides in the register file, the high register is not used.

USP Register Pair

This is where you set the initial address of the user stack. Same remark as above.

6.3.2 Compile Directives and Link Script

With C language[10], if one stack is used, it must reside in data memory. In addition, the C compiler must be invoked without the -mparmusp option and the linker script must specify the libreg9.l and stdreg9.l libraries.

If two stacks are used, the user stack must reside in memory. In addition, you must invoke the C compiler with the option -mparmusp and specify the libureg9.l and stdureg9.l libraries in the linker script.

6.3.3 Start-Up Code

The start-up code includes a routine that copies the initial values of the variables from their storage in ROM to the variables in RAM. Depending on whether you use one or two memory spaces, you must move the data using either the ldpp or lddp instructions.

6.3.4 Summary of Options: Common Cases

6.3.4.1 All in Register File

This is only possible when the program is entirely written in assembler. The C compiler requires that the arguments be passed through a stack located in data memory.

The most important registers to set are MODER and the system stack pointer.

```
ld MODER, #11100000B
;       |||||||+-- HIMP : no foreign access to bus
;       ||||||+--- BRQEN : no foreign access to bus
;       |||+++---- PRS2,1,0 : processor at full speed
;       ||+------- DIV2 : crystal frequency divided by 2
;       |+-------- USP : user stack pointer in registers
;       +--------- SSP : system stack pointer in registers
ld SSPLR, #208; stack starts at end of group 12
ld SSPHR, #255; Unused ; to avoid spurious accesses
```

You can find the corresponding start-up code in the file \program\periph\wdtrect.asm.

The assembly options are the standard ones:

[10] ANOTHER MAJOR CHOICE IS AVAILABLE. IT CONCERNS THE USE OF THE REGISTERS TO PASS THE PARAMETERS. YOU CAN EXCLUDE THE USE OF REGISTERS TO PASS PARAMETERS, THEY ARE THEN ALL PUSHED TO THE STACK. THE COMPILER MUST BE INVOKED WITH THE -NOREGPARM OPTION WITH OR WITHOUT -MPARMUSP. THE LIBRARIES TO USE AT LINK TIME WOULD THEN BE LIBSTK9.L AND STDSTK9.L, OR LIBUSTK9.L AND STDUSTK9.L RESPECTIVELY.

```
GCC9 -c -g -Wa,-alhds <filename>
```

Since we are not using C, no library is linked.

6.3.4.2 Register and Memory, one Stack

If a single stack is used, the C compiler requires that it be located in memory. Initialise the Mode registers and stack pointers as follows:

```
INIT_CIC = Im_gcenm + Im_iamm + 7 ; CICR = IT disabled + Nested Mode + CPL = 7
K_INITCLOCKMODE = MOm_div2m        ; R235 = both stack in memory + clock
                                   ; divided by 2
K_INITWCR = Wm_wdgen          ; WCR = zero wait state + watchdog disabled
Reset:
   spp #0   ; page 0 to access WCR
   ld WCR,#K_INITWCR ; WCR = zero wait state
   ld MODER,#K_INITCLOCKMODE ; init CLOCKMODE (both stack in memory)
   ld CICR,#INIT_CIC ; CIC for our program

   sdm      ; set data memory
   ldw SSPR,#_stack_end ; setup stack
   ldw USPR,#_user_stack_end ; setup user stack (not used actually)
```

Note the sdm instruction that switches all memory accesses to data memory (except the instruction fetches).

You can find the corresponding start-up code in the \program\smcb\startup.asm file.

The C-compiler and the linker will have the following options in their command lines (these lines are taken from the makefile):

```
CFLAGS = -c -g -Wa,-alhds -O -fomit-frame-pointer -Wall
LDFLAGS = -m -I
```

The link script file should specify that the standard libraries be linked:

```
INPUT(\gnu9\bin\stdreg9.l \gnu9\bin\libreg9.l)
```

If you are using initialised variables, a routine must be called to copy the initial values from the ROM to the RAM. This routine belongs to the start-up code. Make sure that it uses the lddp instruction to move the data from program memory to data memory.

6.3.4.3 Register and Memory, one Stack in Memory, one in Registers

When two stacks are used, the first one, the system stack, holds the return addresses. This may reside in registers. The other that holds the arguments and the return values (and also the local variables) must reside in memory. You must initialise the mode registers and stack pointers as follows:

```
INIT_CIC = Im_gcenm + Im_iamm + 7 ; CICR = IT disabled + Nested Mode + CPL = 7
K_INITCLOCKMODE = MOm_div2m + MOm_sspm ; MODER = system stack in register,
   user in memory, clock / 2
K_INITWCR = Wm_wdgen ; WCR = zero wait state + watchdog disabled

Reset:
   ld WCR,#K_INITWCR ; WCR = zero wait state
   ld MODER,#K_INITCLOCKMODE ; init CLOCKMODE
   ld CICR,#INIT_CIC ; CIC for our program

   sdm      ; set data memory
   ld SSPLR,#208 ; setup system stack at end of group C (+1)
   ld SSPHR, #255; Unused ; to avoid spurious accesses
   ldw USPR,#_user_stack_end ; setup user stack
```

Note the sdm instruction that switches all memory accesses to Data Memory (except the instruction fetches).

The corresponding startup code is found in the file \program\barcode\startup.asm.

The C-compiler and the linker will have the following options in their command lines (these lines are extracted from the makefile):

```
CFLAGS = -c -g -Wa,-alhds -O -fomit-frame-pointer -Wall -mparmusp
LDFLAGS = -m -I
```

The link script file should specify that the libraries that use the user stack be linked:

```
INPUT(\gnu9\bin\stdureg9.l \gnu9\bin\libureg9.l)
```

If initialised variables are used, a routine must be called to copy the initial values from the ROM to RAM. This routine belongs to the startup code. Make sure that it uses the lddp instruction to move the data from program memory to data memory.

6.3.4.4 Register and Memory, two Stacks in Memory

This is the same as above, except for the initialisation of the stacks. Mode registers and stack pointers must be initialised as follows:

```
INIT_CIC = Im_gcenm + Im_iamm + 7 ; CICR = IT disabled + Nested Mode + CPL = 7
K_INITCLOCKMODE = MOm_div2m ; R235 = both stack in memory + clock divided by
   2
K_INITWCR = Wm_wdgen ; WCR = zero wait state + watchdog disabled
```

```
Reset:
   ld WCR,#K_INITWCR ; WCR = zero wait state
   ld MODER,#K_INITCLOCKMODE ; init CLOCKMODE (both stack in memory)
   ld CICR,#INIT_CIC ; CIC for our program

   spm      ; set data memory
   ldw SSPR,#_stack_end ; setup stack
   ldw USPR,#_user_stack_end ; setup user stack
```

Note the spm instruction. It confirms that all data are read from the common space, that is from program memory.

The corresponding startup code is found in the file \program\asyncmot\startup.asm.

The C-compiler and the linker will have the following options in their command lines (these lines are extracted from the makefile):

```
CFLAGS = -c -g -Wa,-alhds -O -fomit-frame-pointer -Wall -mparmusp
LDFLAGS = -m -I
```

The link script file should specify that the libraries that use the user stack be linked:

```
INPUT(\gnu9\bin\stdureg9.l \gnu9\bin\libureg9.l)
```

If initialised variables are used, a routine must be called to copy the initial values from ROM to RAM. This routine belongs to the startup code. Make sure that it uses the lddp instruction to move the data from program memory to program memory.

6.4 GLOBAL INITIALISATION: CORE AND PERIPHERALS

6.4.1 Core Initialisation

Set the lower program memory addresses to the addresses of the interrupt service routines that the peripherals will use.

Initialise the core, as shown above.

Initialise the PLL.

If the Multifunction Timers are to work synchronously, you must start them at the same time. To do this,

Reset the Global Counter ENable bit (GCEN, bit 7 of register CICR) before the MFTs are initialised.

Then set it just before the main program starts.

In the examples above, the GCEN is set, as this is the default value on reset. If it is necessary to synchronise the timers, the line:

`INIT_CIC = Im_gcenm + Im_iamm + 7 ; CICR = IT disabled + Nested Mode + CPL = 7`
should read:

`INIT_CIC = Im_iamm + 7 ; CIC = IT & Counters disabled + Nested Mode + CPL = 7`
Initialise the memory. This may include:

Resetting the register file and/or the data memory to zero, and

Preloading the variables with the initial values.

The procedure to follow when programming in C is described above.

6.4.2 Peripheral Initialisation

It is now time to configure the peripherals. For each one, you must take the following steps:

Set the Page Pointer Register to the page that contains the peripheral's registers. On some peripherals, several pages are involved so set the Page Pointer Register accordingly.

Set the various registers that define the behaviour of the peripherals. In some cases, you have to set them in a certain order. Refer to the appropriate Data Sheet. Do not yet set the bits that start the peripheral working.

Set the Interrupt Vector Register of each peripheral to point to the corresponding group of pointers to the interrupt service routines in program memory.

You can now set the Interrupt Mask Register, since the global Interrupt ENable bit is reset.

If you are using DMA, set the DAPR and DCPR registers to point to two registers in the register file. Then:

Initialise the address register to the address of the buffer in which the data is to be stored.

Set the counter register to zero to inhibit DMA transfers.

You can now set the peripheral Enable bit or Start bit, unless you only want it to start later (as can be the case with the Watchdog Timer).

6.4.3 Port Initialisation

When you have configured all the peripherals as just described, you should initialise the ports to your requirements: input or output, open drain or push-pull, TTL or CMOS levels, etc. for the parallel I/O ports.

You must set the port pins that serve as inputs or outputs for peripherals as either

– Alternate Function for peripheral outputs, or

– Regular Input for peripheral inputs

The exception is the A to D converter, where you must set the input pin as Alternate Function.

6.4.4 Final Initialisation

Install the Working Registers in the group defined for the main program.

Set the Page Pointer Register to a default value.

Set the Global Counter Enable, the global Interrupt ENable and, if required the Watchdog function.

The main program is now ready to run.

6.5 INTERRUPT CONSIDERATIONS

The ST9+ has two main paging registers:

– The RPP register pair that selects the working register group

– The PPR that selects one of 256 pages within register group 15

These registers are essential to the correct working of the program.

When an interrupt occurs, it is likely you will have to change either or both of these registers. This is why you must push them to the stack on entry, and pop them back on exit.

An interrupt is requested by a bit in a register of a peripheral. This bit is not reset automatically by the fact that the interrupt is serviced. You must reset it in the code of your interrupt service routine.

If you write an interrupt service routine in C, by default all registers used by the routine are pushed on the stack. This can be time-consuming. If your program is entirely written in C, and you use none or few registers explicitly, the register file has enough room to allocate a private working register group for each interrupt service routine. You specify this by a #pragma pseudo-instruction. This method provides the fastest response times.

6.6 C-LANGUAGE AND ASSEMBLER INTERFACING

The C compiler offers a powerful language for defining assembly-language functions within C source text. This language allows you to specify the arguments and their types, and the type of the return value. Please refer to the C compiler manual.

You may often find it useful to write a routine within an assembler source file, and make it global, so that it can be called from a C source text. In this case, take care that the routine stays in accordance with C conventions.

Arguments are passed in registers if possible.

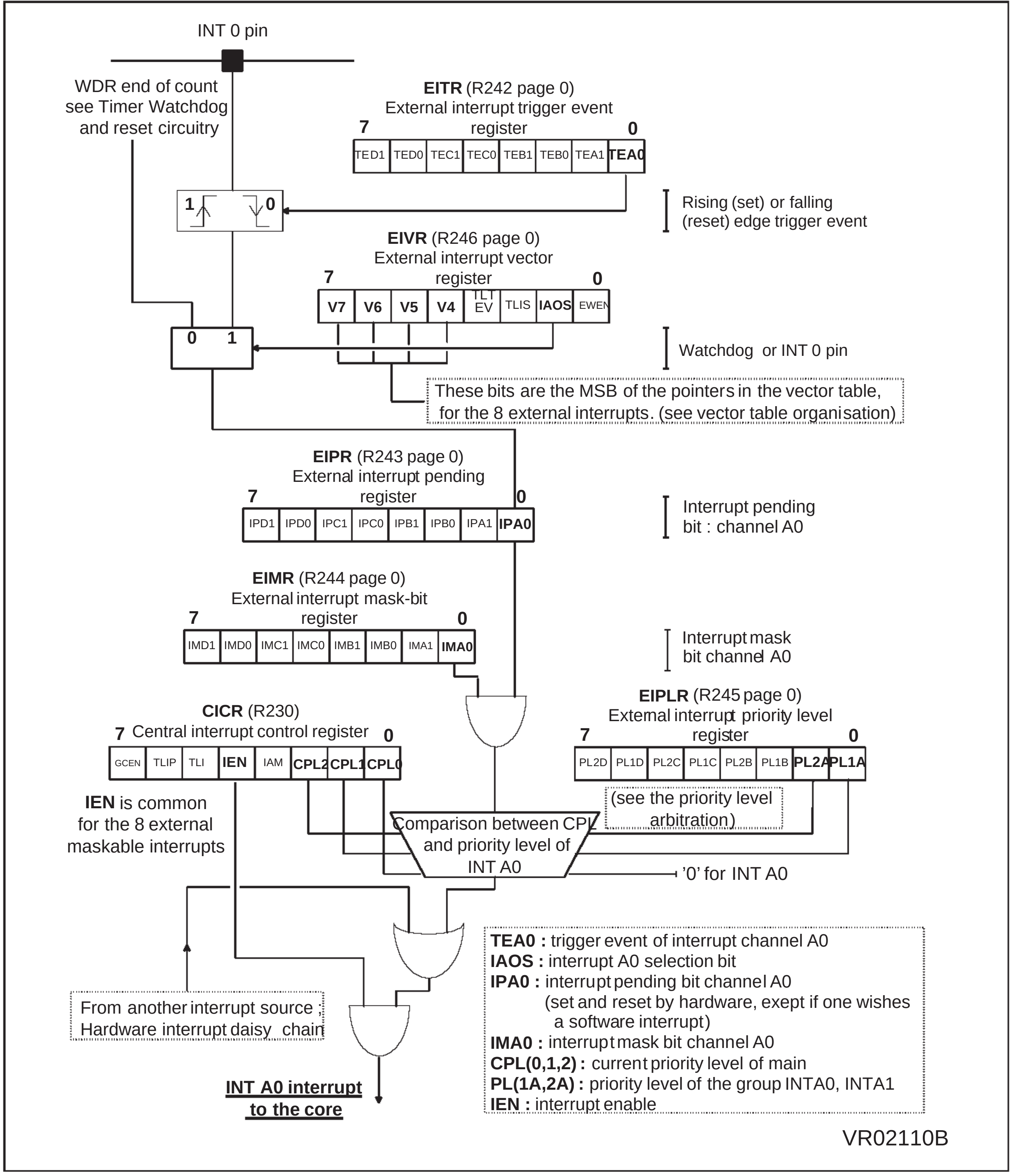
It is important to know that the only registers that may be changed by the assembler function in the current working register group are rr0, rr2 and rr4. Don't touch the other registers or your program may crash. If you need more registers, push the other registers on the stack. You could also change the working register group to a private group, but the original group must be restored on exit (before setting the registers holding the return value).

7 DETAILED BLOCK DIAGRAMS

Here are the detailed block diagrams to supplement to the simplified ones used at various points throughout this book.

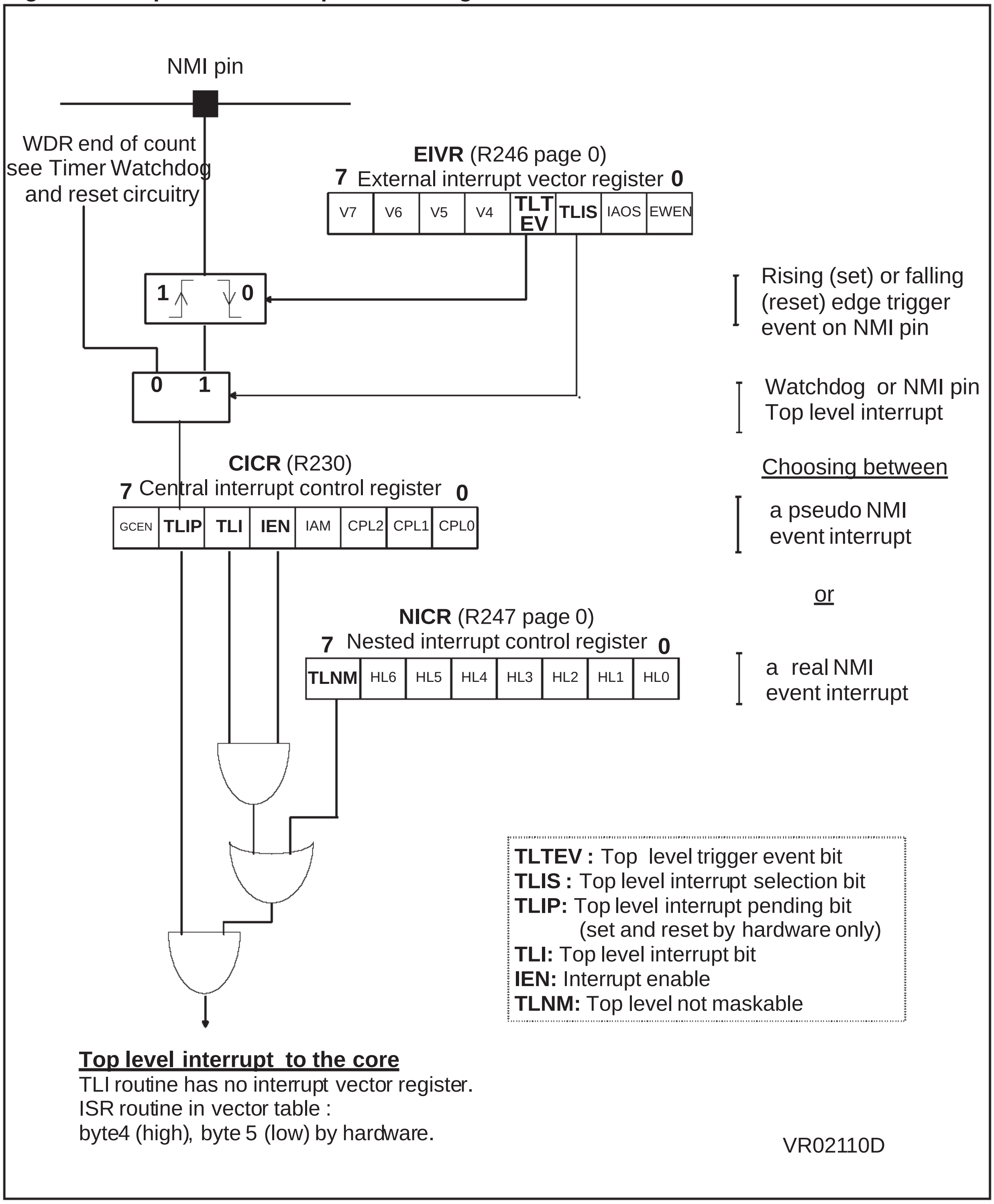
7.1 EXTERNAL INTERRUPT CONTROLLER

Figure 47. External Interrupt Block diagram


INT 0 pin
WDR end of count see Timer Watchdog and reset circuitry
EITR (R242 page 0) External interrupt trigger event register
7
0
TED1 TED0 TEC1 TEC0 TEB1 TEB0 TEA1 TEA0
1
0
Rising (set) or falling (reset) edge trigger event
EIVR (R246 page 0) External interrupt vector register
7
0
V7 V6 V5 V4 TLTEV TLIS IAOS EWEN
0 1
Watchdog or INT 0 pin
These bits are the MSB of the pointers in the vector table, for the 8 external interrupts. (see vector table organisation)
EIPR (R243 page 0) External interrupt pending register
7
0
IPD1 IPD0 IPC1 IPC0 IPB1 IPB0 IPA1 IPA0
Interrupt pending bit : channel A0
EIMR (R244 page 0) External interrupt mask-bit register
7
0
IMD1 IMD0 IMC1 IMC0 IMB1 IMB0 IMA1 IMA0
Interrupt mask bit channel A0
CICR (R230) Central interrupt control register
7
0
GCEN TLIP TLI IEN IAM CPL2 CPL1 CPL0
EIPLR (R245 page 0) External interrupt priority level register
7
0
PL2D PL1D PL2C PL1C PL2B PL1B PL2A PL1A
(see the priority level arbitration)
IEN is common for the 8 external maskable interrupts
Comparison between CPL and priority level of INT A0
'0' for INT A0
From another interrupt source , Hardware interrupt daisy chain
INT A0 interrupt to the core
TEA0 : trigger event of interrupt channel A0
IAOS : interrupt A0 selection bit
IPA0 : interrupt pending bit channel A0 (set and reset by hardware, exept if one wishes a software interrupt)
IMA0 : interrupt mask bit channel A0
CPL(0,1,2) : current priority level of main
PL(1A,2A) : priority level of the group INTA0, INTA1
IEN : interrupt enable
VR02110B

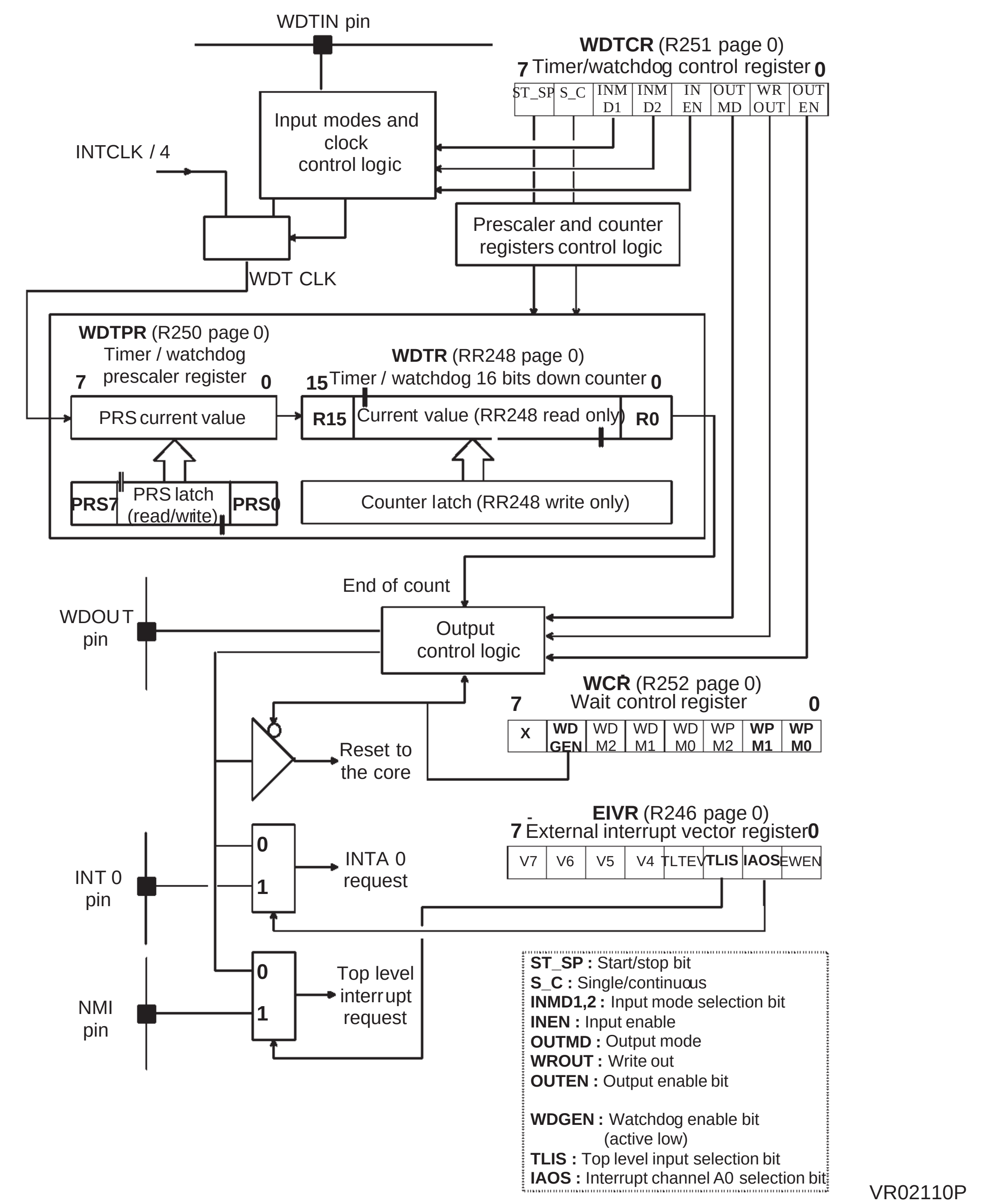
7.2 TOP-LEVEL INTERRUPT INPUT

Figure 48. Top-Level Interrupt Block Diagram


NMI pin
WDR end of count
see Timer Watchdog
and reset circuitry
EIVR (R246 page 0)
7 External interrupt vector register 0
V7
V6
V5
V4
TLT EV
TLIS
IAOS
EWEN
Rising (set) or falling (reset) edge trigger event on NMI pin
1
0
0
1
Watchdog or NMI pin Top level interrupt
Choosing between
a pseudo NMI event interrupt
or
a real NMI event interrupt
CICR (R230)
7 Central interrupt control register 0
GCEN
TLIP
TLI
IEN
IAM
CPL2
CPL1
CPL0
NICR (R247 page 0)
7 Nested interrupt control register 0
TLNM
HL6
HL5
HL4
HL3
HL2
HL1
HL0
TLTEV : Top level trigger event bit
TLIS : Top level interrupt selection bit
TLIP: Top level interrupt pending bit (set and reset by hardware only)
TLI: Top level interrupt bit
IEN: Interrupt enable
TLNM: Top level not maskable
Top level interrupt to the core
TLI routine has no interrupt vector register.
ISR routine in vector table :
byte4 (high), byte 5 (low) by hardware.
VR02110D

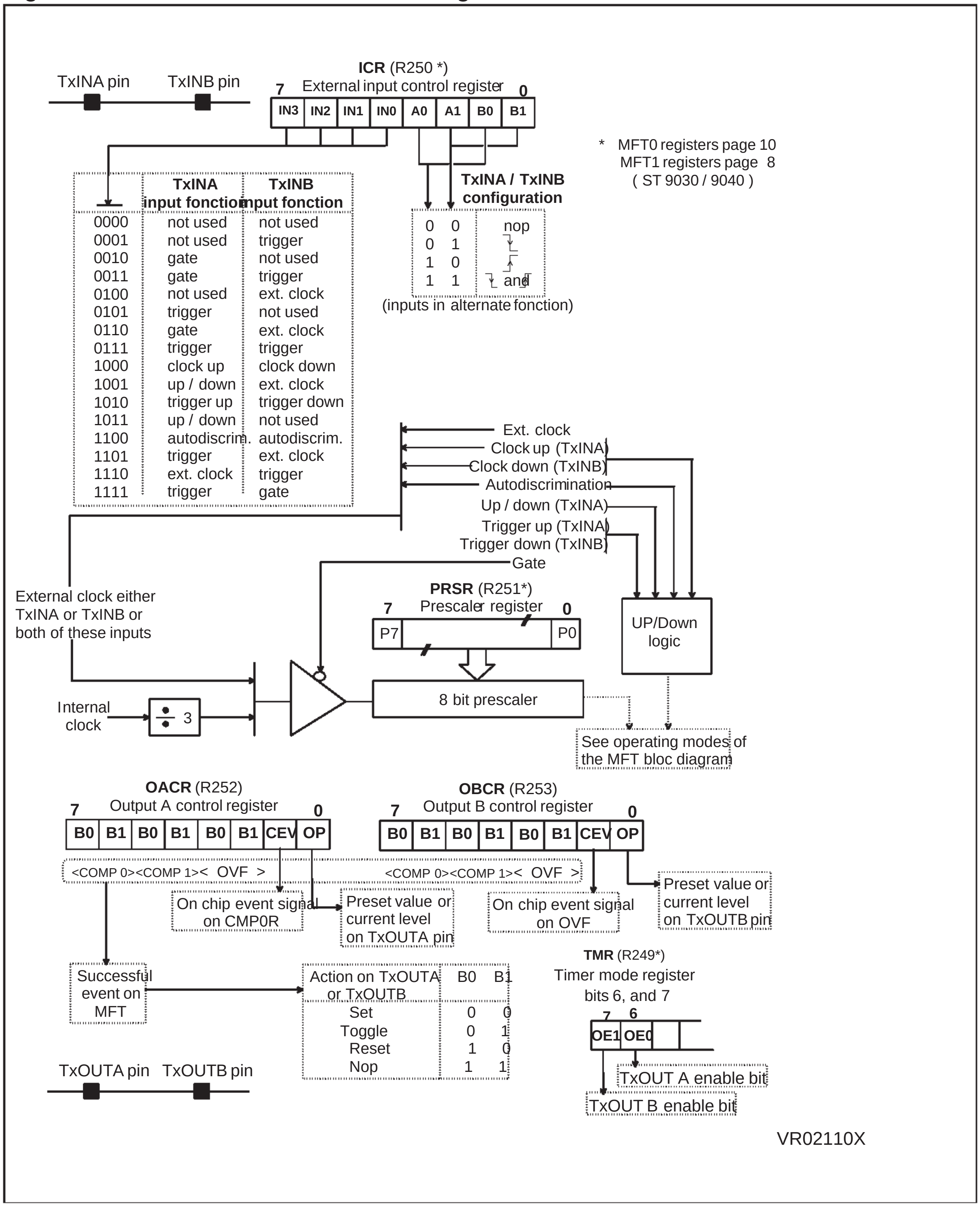
7.3 WATCHDOG TIMER

Figure 49. Watchdog Timer Block Diagram


WDTIN pin
WDTCR (R251 page 0)
7 Timer/watchdog control register 0
ST_SP
S_C
INMD1
INMD2
INEN
OUTMD
WROUT
OUTEN
Input modes and clock control logic
INTCLK / 4
Prescaler and counter registers control logic
WDT CLK
WDTPR (R250 page 0) Timer / watchdog prescaler register
7
0
WDTR (RR248 page 0)
15 Timer / watchdog 16 bits down counter 0
PRS current value
R15
Current value (RR248 read only)
R0
PRS7
PRS latch (read/write)
PRS0
Counter latch (RR248 write only)
End of count
WDOUT pin
Output control logic
WCR (R252 page 0)
7 Wait control register 0
X
WDGEN
WDM2
WDM1
WDM0
WPM2
WPM1
WPM0
Reset to the core
EIVR (R246 page 0)
7 External interrupt vector register 0
V7
V6
V5
V4
TLTEV
TLIS
IAOS
EWEN
INT 0 pin
0
1
INTA 0 request
NMI pin
0
1
Top level interrupt request
ST_SP : Start/stop bit
S_C : Single/continuous
INMD1,2 : Input mode selection bit
INEN : Input enable
OUTMD : Output mode
WROUT : Write out
OUTEN : Output enable bit
WDGEN : Watchdog enable bit (active low)
TLIS : Top level input selection bit
IAOS : Interrupt channel A0 selection bit
VR02110P

7.4 MULTIFUNCTION TIMER

Figure 50. Multifunction Timer Block Diagram



	TxINA input fonction	TxINB input fonction
0000	not used	not used
0001	not used	trigger
0010	gate	not used
0011	gate	trigger
0100	not used	ext. clock
0101	trigger	not used
0110	gate	ext. clock
0111	trigger	trigger
1000	clock up	clock down
1001	up / down	ext. clock
1010	trigger up	trigger down
1011	up / down	not used
1100	autodiscrim.	autodiscrim.
1101	trigger	ext. clock
1110	ext. clock	trigger
1111	trigger	gate

0	0	nop
0	1	↓
1	0	↑
1	1	↓ and ↑

Action on TxOUTA or TxOUTB	B0	B1
Set	0	0
Toggle	0	1
Reset	1	0
Nop	1	1

Figure 51. MFT Operating Modes Block Diagram

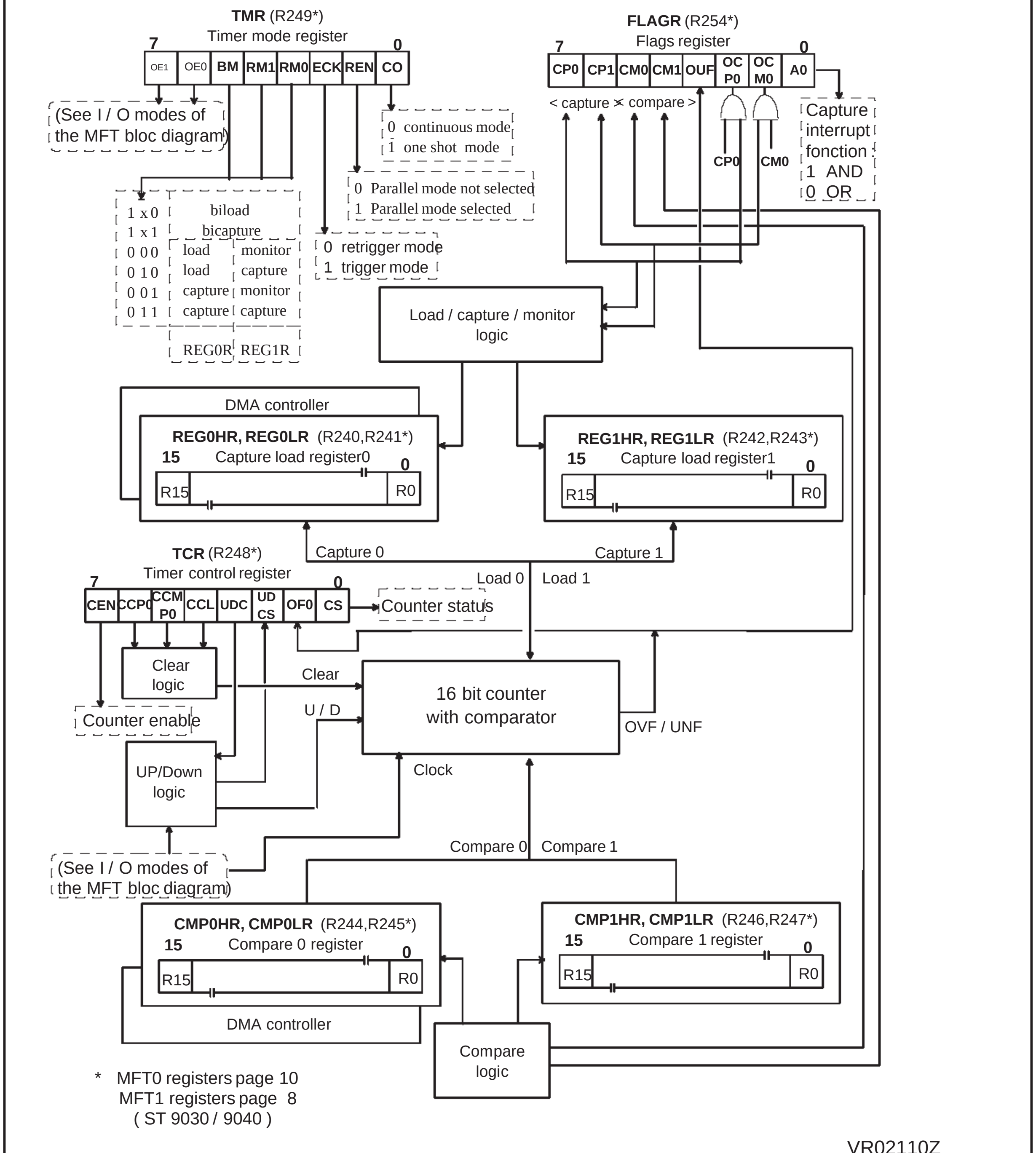
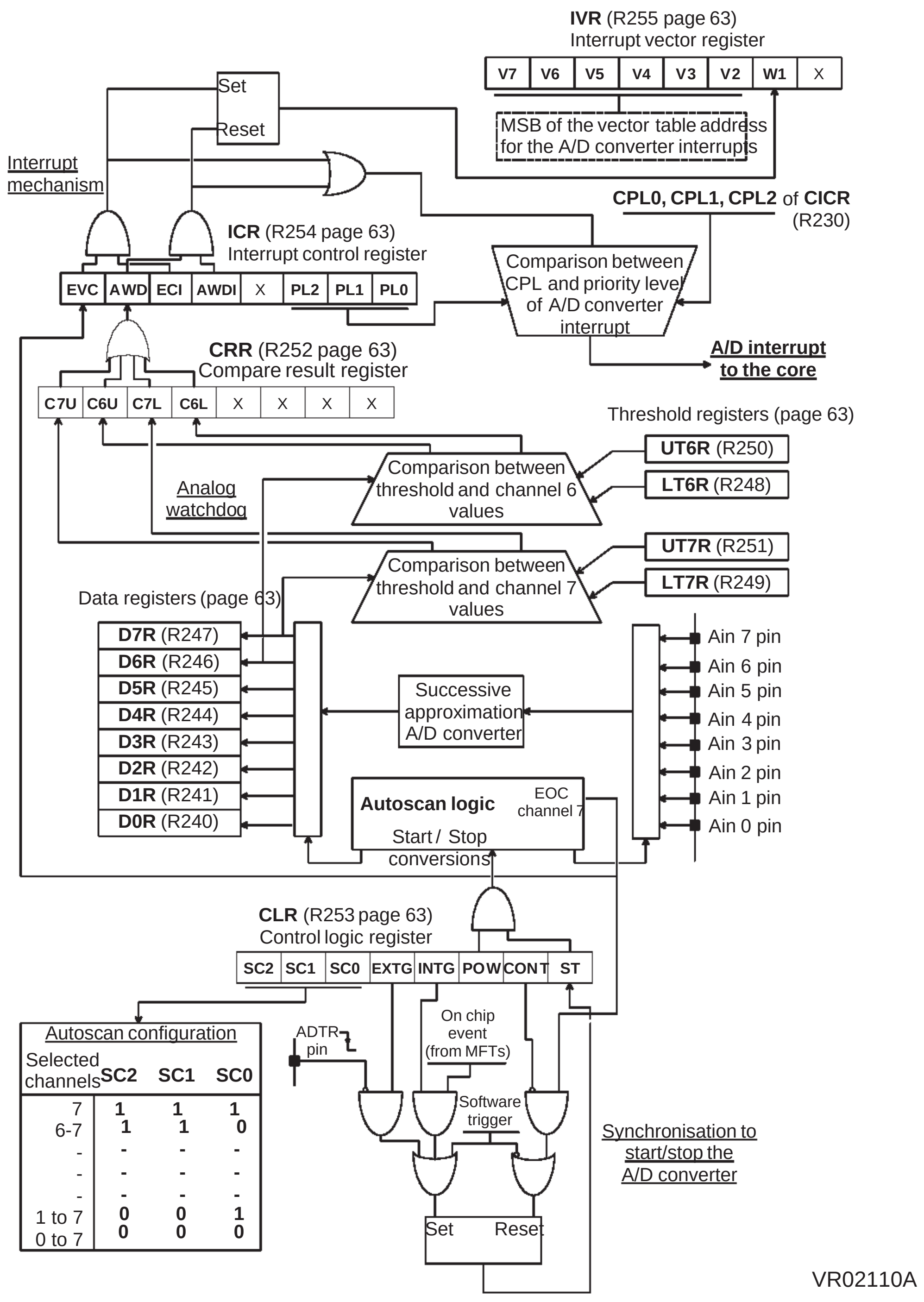


Figure 52. Analog to Digital Converter Block Diagram



8 GLOSSARY

A

A/D	Analog to Digital
AC	Alternating Current
ACR	Address Compare Register
ADC	Analog to digital converter
ADTR	A/D Trigger
AIN	Analog input
AWD	Analog watchdog

B

BRG	Bit Rate Generator

C

CCU	Clock Control Unit
CHCR	Character Configuration Register
CICR	Central Interrupt Control Register
CLK	Clock
CLR	Control Logic Register
CMOS	Complementary Metal Oxide Silicon
CMP	Compare Register
CPL	Current Priority Level
CR	Carriage Return
CRR	Compare Results Register
CS	Chip Select

D

DAPR	DMA Address Pointer Register
DC	Direct Current
DCPR	DMA Counter Pointer Register
DI	Data In
DIL	Dual In line
DMA	Direct Memory Access
DO	Data Out
DP	Data Page

E

ECV	End of conversion
EEPROM	Electrically-Erasable PROM
EIMR	External Interrupt Mask-Bit Register
EIPLR	External Interrupt Priority Level Register
EIPR	External Interrupt Pending Register
EITR	External Interrupt Trigger Register
EIVR	External Interrupt Vector Register
EWDS	Erase or Write Disable
EWEN	Erase or Write Enable

F

FLAGR	Flag Register

GLOSSARY

G

GCC9	GNU C Compiler for ST9+

I

I/O	Input/Output
IAM	Interrupt Arbitration Mode
ICR	External Input Control Register
IDCR	Interrupt/ DMA Control Register
IDMR	Interrupt/ DMA Mask Register
IDPR	Interrupt/DMA Priority Register
IEN	Interrupt Enable
INMD	Interrupt Mode
INT	Interrupt
IOCR	I/O Connection Register
ISR	Interrupt Segment Register or Interrupt Service Routine
IVR	Interrupt Vector Register

L

LCD	Liquid Crystal Display
LED	Light Emitting Diode
LF	Line Feed
LSB	Least Significant Bit

M

MFT	Multifunction Timer
MMU	Memory Management Unit
MODER	Mode Register
MSB	Most Significant Bit

N

NICR	Nested Interrupt Control Register
NMI	Non-Maskable Interrupt

O

OACR	Output A Control Register
OBCR	Output A Control Register
OVF	Overflow

P

PC	Program Counter or Personal Computer
PFE	Programmer’s File Editor
PPR	Page Pointer Register
PRL	Priority Level
PROM	Programmable ROM
PWM	Pulse Width Modulator
PXC	Port Control Register
PXDR	Port Data Register

R

RAM	Random Access Memory
RCCU	Reset and Clock Control Unit
ROM	Read-Only Memory
RPP	Working Register Pointer
RXCLK	Receiver Clock Input

RXDATA	Receiver Data

S

SEG	Extract the 6 bits segment of a label
SCI	Serial Communications Interface
SCK	Serial Clock
SOF	Extract the 16 bits offset of a label
SPI	Serial Peripheral Interface
SPICR	SPI Control Register
SPIDR	SPI Data Register
SRP	Set Register Pointer
SSPR	System Stack Pointer Register

T

TCR	Timer Control Register
TLNM	Top Level Not Maskable
TMR	Timer Mode Register
TTL	Transistor to Transistor Logic
TXCLK	Transmitter Clock Input
TXD	Transmitter Data

U

U/D	Up/Down
UART	Universal Asynchronous Receiver/Transmitter
UNF	Underflow
USPR	User Stack Pointer Register
UV	Ultraviolet

W

WDT	Watchdog Timer
WDTCR	Watchdog Timer Control Register
WDTLR	Watchdog Timer Low Register
WDTPR	Watchdog Timer Prescaler Register

9 INDEX

Index

Index

Index

Index