



AN1084 APPLICATION NOTE

GETTING STARTED WITH THE ST92141 SOFTWARE LIBRARY VERSION 1.0

by V. Onde/Microcontroller Division Applications

INTRODUCTION

This Application Note describes the software library developed for the ST92141 MCU. This 8/16-bit microcontroller contains a cell dedicated to 3-phase sine wave generation, making it suitable for standard single- and three-phase AC motor drives and uninterruptible power supplies (UPS).

The current library consists of several C modules that contain a set of convenient functions for the scalar control of AC motors.

It will allow users to quickly evaluate both MCUs and available tools, and to have a motor running in less than a week using a standard 3-phase power inverter. It also eliminates the need for developing time consuming sine wave generation software by providing ready-to-use functions that let the user concentrate on his application layer.

A basic knowledge of C programming, AC motor drives and power inverter hardware is required. In-depth know-how of ST92141 functions is only required for customizing delivered modules and when adding new ones (grey modules in Figure 1.) for a complete application platform.

The Version 1.0 Software Package can be obtained from an ST Sales Office. In regards to Version 2.0, most of modules, tools and MCU issues described below remain valid. Enhancement and upgrade issues related to Version 2.0 are described in application note AN1277.

Table of Contents

INTRODUCTION	1
1 GETTING STARTED WITH TOOLS	7
1.1 INSTALLATION	7
1.1.1 Toolchain	7
1.1.2 EPROMer	7
1.1.3 Technical Literature	8
1.2 COMPILING THE LIBRARY	9
1.3 CUSTOMIZING THE MAKEFILE	11
1.3.1 Important Information	11
1.3.2 Adding Source Files to your Project	11
1.3.3 Modifying Compile Options	11
1.3.4 Dependencies	11
2 ST9+ CORE SPECIFICS	12
2.1 MEMORY AND REGISTER FILE MAPPING	12
2.1.1 ACMOTOR.SCR	12
2.1.2 RAM: Variables & User Stack	12
2.1.3 REGISTER FILE	13
2.1.3.1 Mapping	13
2.1.3.2 Customization	13
2.2 DEFINE.H	14
2.3 CRT9.ASM FILE	14
3 PRESENTATION OF MODULES AND LIBRARY FUNCTIONS	15
3.1 LIBRARY REFERENCES	15
3.2 IMC MODULE	16
3.2.1 Description	16
3.2.2 Interrupts	16
3.2.2.1 ADT_IT	16
3.2.2.2 ZPC_IT	17
3.2.2.3 CPT_IT	17
3.2.2.4 OTC_IT	18
3.2.3 List of Available Functions	18
3.2.4 Detailed Explanations and Customization	33

Table of Contents

3.2.4.1	ST92141 Library Main Timebase	33
3.2.4.2	Software Timebase	34
3.2.4.3	IMC Software Watchdog	35
3.2.4.4	IMC_GetRotorFreq	35
3.2.4.5	Customizing Rotor Frequency Acquisition	37
3.2.4.6	Tacho Compare Event Issues	38
3.3	ACMOTOR MODULE	38
3.3.1	Description	38
3.3.2	List of Available Functions	39
3.3.3	Detailed Explanations and Customization	56
3.3.3.1	ACM_VoltageMaxAllowed	56
3.3.3.2	ACM_GetOptimumSlip	57
3.3.3.3	ACM_SlipRegulation	58
3.3.3.4	ACM_SoftStart	59
3.3.3.5	Using P Pole Motors	60
3.3.3.6	Stator Frequencies above 340 Hz	61
3.4	ADC MODULE	62
3.4.1	Purpose	62
3.4.2	Description	62
3.4.3	Synopsis	62
3.4.4	Memory Use	63
3.4.5	Timings	63
3.4.6	Software Watchdog	63
3.4.7	Caution	64
3.4.7.1	ADC Register Declaration File Version	64
3.4.7.2	Sampling Issues	64
3.4.7.3	Value Availability Time	64
3.4.7.4	Interrupts	64
3.4.7.5	Port Initialization	64
3.4.8	Customizing the ADC Module	64
3.5	FUZZY LOGIC REGULATION	66
3.5.1	Description	66
3.5.2	Fuzzy Engine Technical Characteristics	67
3.5.3	Customization	67
3.5.3.1	Updating the ftc8.I Fuzzy Kernel Library	67
3.5.3.2	Modifying the Fuzzy Engine	68

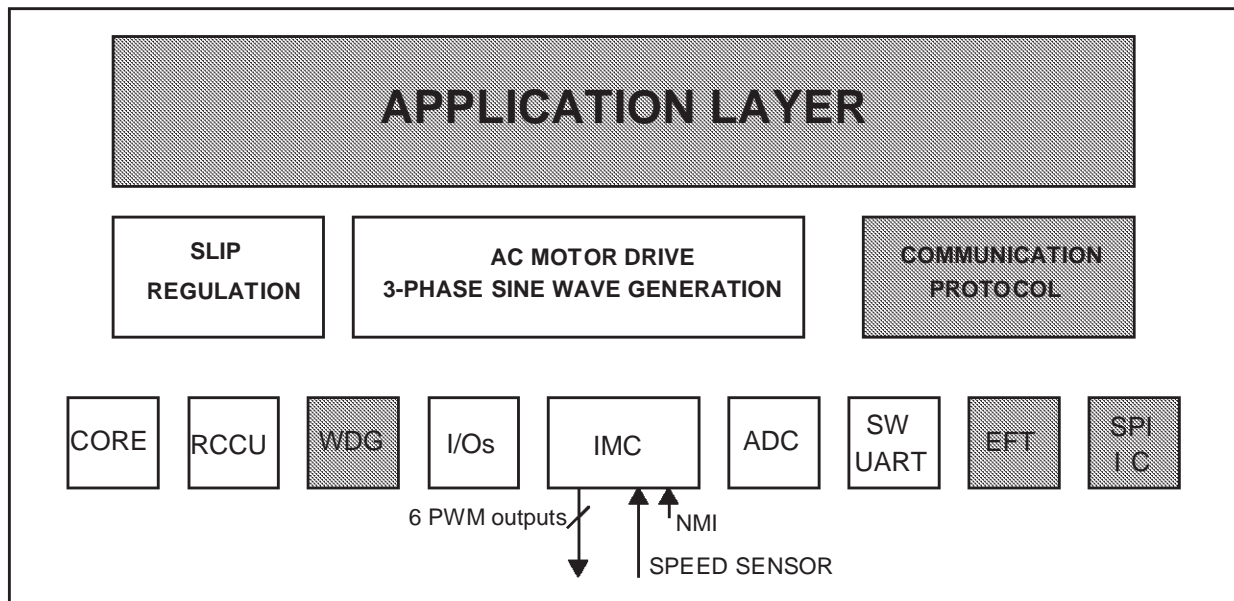
Table of Contents

3.6 I/O MODULE	69
3.6.1 Description	69
3.6.2 I/O Access	69
3.7 RCCU MODULE	69
3.7.1 Description	69
3.7.2 Customization	69
3.8 CORE MODULE	70
3.8.1 Description	70
3.8.2 Available Functions	70
3.8.3 Customization	70
3.8.3.1 DIV_Zero_Trap	70
3.8.3.2 External Interrupts	70
3.8.3.3 NMI Interrupt	70
3.8.4 UART Module	71
3.8.5 Description	71
3.8.6 ST92141 Characteristics	71
3.8.7 PC Characteristics	72
3.8.8 Customization	72
3.8.9 Important Notice for Hardware Implementation	72
3.9 CODE EXAMPLE	73
3.9.1 Open Loop: Voltage and Frequency are individually adjustable	73
3.9.2 Closed Loop	73
3.10 DESIGNING WITH 92141LIB: LIBRARY INTEGRATION	75
4 APPENDIX	77
4.1 IMC MODULE FLOWCHARTS	77
4.1.1 Automatic Data Transfer Interrupt (ADT)	77
4.1.2 Zero of PWM Counter Interrupt (ZPC)	79
4.1.3 Tacho Capture Interrupt (CPT)	80
4.1.4 IMC_GetRotorFreq	81
4.2 ACMOTOR MODULE FLOWCHARTS	82
4.2.1 ACM_SoftStart	82
4.2.2 ACM_RampUp	83
4.2.3 ACM_Update_Sine_Tables	84
4.2.4 ACM_SlipRegulation	85

Table of Contents

4.3 TOOL OPTION SUMMARY	86
4.3.1 Linker	86
4.3.2 Assembler	87
4.3.3 C Compiler (GCC9)	88
4.4 CREATE YOUR OWN FTC8.L FILE	91

Figure 1. Overall Software Architecture



ST92141 Library Version 1.0 Characteristics (CPU running at 25 MHz):

- Stator Frequency Range: From 1.0 Hz up to 680.0 Hz with resolution greater than 0.5 Hz,
- Voltage Resolution: 8-bit modulation index,
- 10-bit PWM Generation
- PWM Frequency: Greater than or equal to 12.2 kHz, variable above 27.5 Hz, centred mode,
- CPU Load (sine wave generation only) less than 15%, depending on the output frequency. (Due to IMC's ADT interrupt, the CPU load will be proportional to the output frequency. For information, it will only be 3% @ 60 Hz, 6% @ 127 Hz, 13% @ 254 Hz and < 15% above 254 Hz.)
- Required ROM: < 5.5 Ko including fuzzy regulation code,
- Required RAM: 370 octets.

Note: These figures are for information only; this software library may be subject to changes to improve performances depending on the use of the final application and peripheral resources. It was build using robustness-oriented structures as much as possible, therefore preventing the speed or density from being optimized.

The 12.2 kHz switching frequency is used to take advantage of all IMC compare register capabilities, thus providing a real 10-bit PWM with a 25-MHz CPU clock. In addition, this frequency is a good compromise between the reduction of switching losses and acoustic noise (rejected in the inaudible range due to centred mode PWM patterns).

1 GETTING STARTED WITH TOOLS

1.1 INSTALLATION

1.1.1 Toolchain

This library has been compiled using the January 2000, 4.3.3 release of the ST9+ Software Toolchain, which can be found on the MCU CD-ROM or downloaded from the ST website (<http://www.st.com>, in the Microcontroller section, then in ST9/Development Tools/Free Software).

It consists of the GNU C Compiler (GCC9) interfaced with the TR9 Macro-expander, the GAS9 Assembler and the LD9 Linker, plus other tools and utilities. A valid license is required to use the GNU C Compiler. A GDB9 Debugger, coupled with a Windows graphical interface (WGDB9) is also provided for working with the ST92141 hardware emulator.

All components must be installed, with the exception of ST Visual Make.

When installed, the ST92141 Library can be copied locally (i.e. in `c:\code\st92141`). To complete the set-up routine, the user must then change the properties of the "Toolchain DOS Session" short-cut found in the "ST9+ Toolchain-Jan. 2000" folder of the start menu. The working directory name must be changed from `\toolst9p\Examples\st92r195\` to the path name of the library (i.e. `\code\st92141\` using the previous example). Before starting, make sure that:

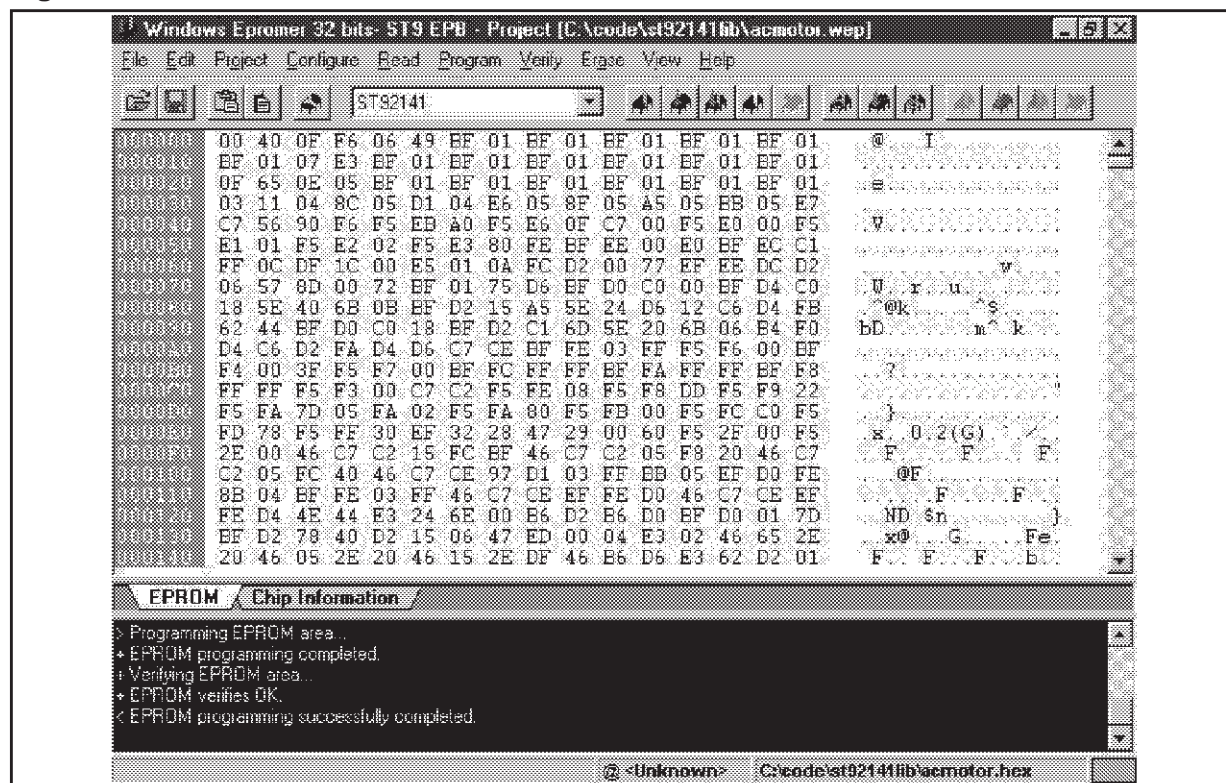
- a `c:\temp` directory exists on the hard disk, otherwise it must be created (required for the C compiler),
- the `\toolst9p\include.st9\` file contains release 3.0 of the `imc.h` file (IMC peripheral register declaration), or else the version included in the library (in the "include" directory).

1.1.2 EPROMer

In order to burn an EPROM with the generated hexadecimal file (Intel .hex format), the user should also install the STVP (ST Visual Programmer).

The ST Visual Programmer is the common standard EPROMer interface used for ST Microcontroller Programming Systems (EPROM Programming Boards - EPBs). This tool is used to read, program, verify and check ST92E141(EPROM) or ST92T141(OTP) MCUs.

Figure 2. ST Visual EPROMer



1.1.3 Technical Literature

Technical literature is also available on the ST website in the Microcontroller section. These documents are useful for better understanding ST9+ MCUs.

ST9+ Software Library/User Guide Companion Software is also available on the ST website under ST9 / Development Tools / Free Software.

The ST9+ User Guide is located under ST9 / Technical Documents / User Guide.

1.2 COMPILING THE LIBRARY

Compiling is achieved using the GMAKE utility tool. This tool is executed in the Toolchain DOS session window. All options and file declarations are defined inside a makefile file included in the library.

The first step is to regenerate dependencies which are listed in a makedep file. This is done by typing the following instruction at the DOS prompt (following the previous example):

```
c:\code\st92141> gmake -k dep
```

Note that the original makedep file is delivered empty (required for the GMAKE utility).

Dependencies will be listed in the makedep file generated in the working directory.

Once this is done, the next step is to re-compile the source files by typing:

```
c:\code\st92141> gmake
```

When the compilation flow is achieved successfully, a .u extension file (acmotor.u in the current example) is generated. If unsuccessful, an error message will be displayed on the last line and an .u file will not be present. If this occurs using the delivered source, make sure that a c:\temp directory exists on the hard disk (required for the C compiler).

This acmotor.u file can be directly used by the debugger (it contains all necessary debugging information).

The last step is to generate an executable file used for burning an EPROM or OTP, type:

```
c:\code\st92141> gmake intel
```

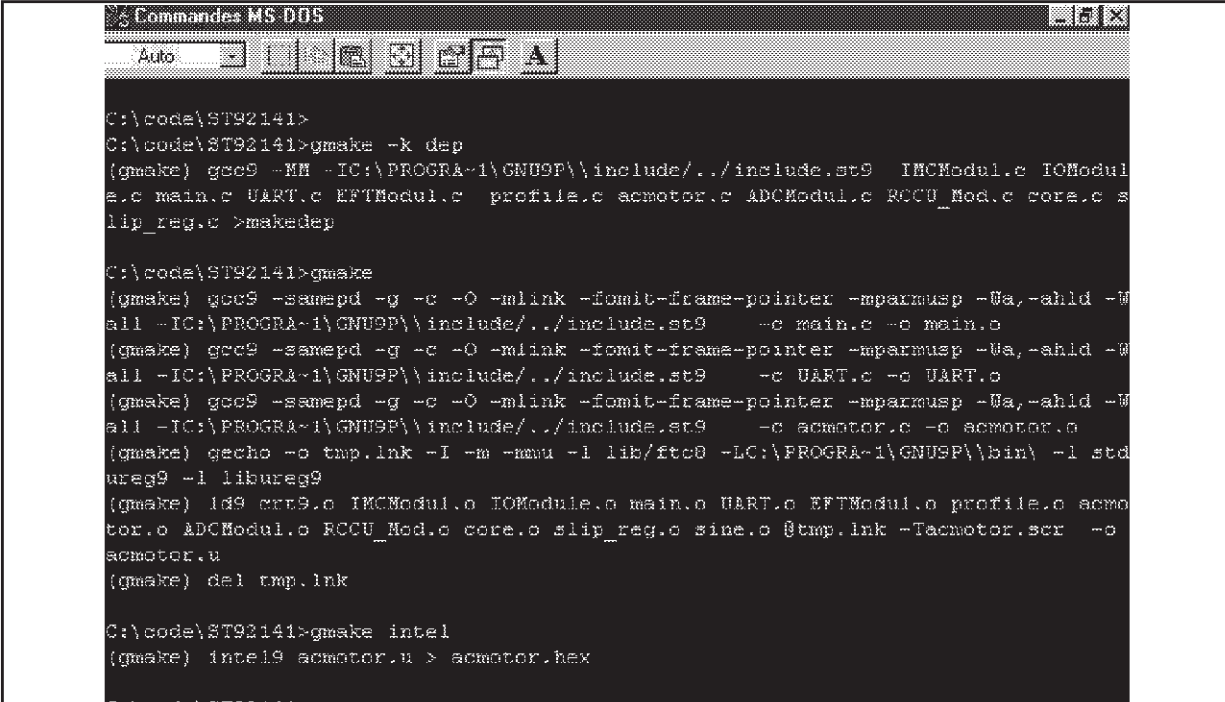
This will produce an executable file (.hex intel format) from the previous .u file.

To clean-up your working directory (i.e. delete all files except source and initialization files), type:

```
c:\code\st92141> gmake clean
```

Some of these command lines are summarized in Figure 3.. Please note that the Verbose option of the linker was disabled in order to obtain a screen shot that includes three different commands (gmake -k dep, gmake and gmake intel).

Figure 3. DOS Toolchain Session Interface



```
C:\code\ST92141>
C:\code\ST92141>gmake -k dep
(gmake) gcc9 -MM -IC:\PROGRA~1\GNU9P\include\../include.st9 IMCModul.o IOModul
e.o main.o UART.o EFTModul.o profile.o acmotor.o ADCModul.o RCCU_Mod.o core.o s
lip_reg.o >makedep

C:\code\ST92141>gmake
(gmake) gcc9 -samedp -g -c -O -mink -fomit-frame-pointer -mparmusp -Wa,-ahld -W
all -IC:\PROGRA~1\GNU9P\include\../include.st9 -c main.o -o main.o
(gmake) gcc9 -samedp -g -c -O -mink -fomit-frame-pointer -mparmusp -Wa,-ahld -W
all -IC:\PROGRA~1\GNU9P\include\../include.st9 -c UART.o -o UART.o
(gmake) gcc9 -samedp -g -c -O -mink -fomit-frame-pointer -mparmusp -Wa,-ahld -W
all -IC:\PROGRA~1\GNU9P\include\../include.st9 -c acmotor.o -o acmotor.o
(gmake) gcc9 -o tmp.lnk -I -m -mmu -l lib/ftc0 -LC:\PROGRA~1\GNU9P\bin\ -l std
ureg9 -l libureg9
(gmake) ld9 crt9.o IMCModul.o IOModule.o main.o UART.o EFTModul.o profile.o acmo
tor.o ADCModul.o RCCU_Mod.o core.o slip_reg.o sine.o @tmp.lnk -Tachmotor.scr -o
achmotor.u
(gmake) del tmp.lnk

C:\code\ST92141>gmake intel
(gmake) intel9 achmotor.u > achmotor.hex
```

1.3 CUSTOMIZING THE MAKEFILE

1.3.1 Important Information

Some code editors may replace tabulations with spaces when editing the makefile. Therefore, the makefile must be edited within the DOS session using the EDIT utility, or otherwise the GMAKE utility will fail.

1.3.2 Adding Source Files to your Project

C sources are listed after the Clist_SRC symbol in the makefile and are space-separated. Assembly files are listed after ASMlist_SRC, with the exception of the crt9.asm file, which must be linked at the beginning of the ROM area (contains reset vector, etc.).

1.3.3 Modifying Compile Options

Options are applied to the tools launched during the make process: the GCC9 is used for pre-processing, compiling and assembling, the LD9 is used for linking.

They are listed in the makefile and defined as CFLAGS, ASMFLAGS and LDFLAGS.

Further details on selected options can be found in the appendix.

1.3.4 Dependencies

The makedep file containing the dependencies must be re-generated each time they are modified. This is done by typing *gmake -k dep* in the Toolchain DOS session.

2 ST9+ CORE SPECIFICS

2.1 MEMORY AND REGISTER FILE MAPPING

Since the addressed memory area in the ST92141 MCU is less than 64 Kbytes, the Memory Management Unit (MMU) registers (initialized in crt9.asm) do not need to be changed during run-time.

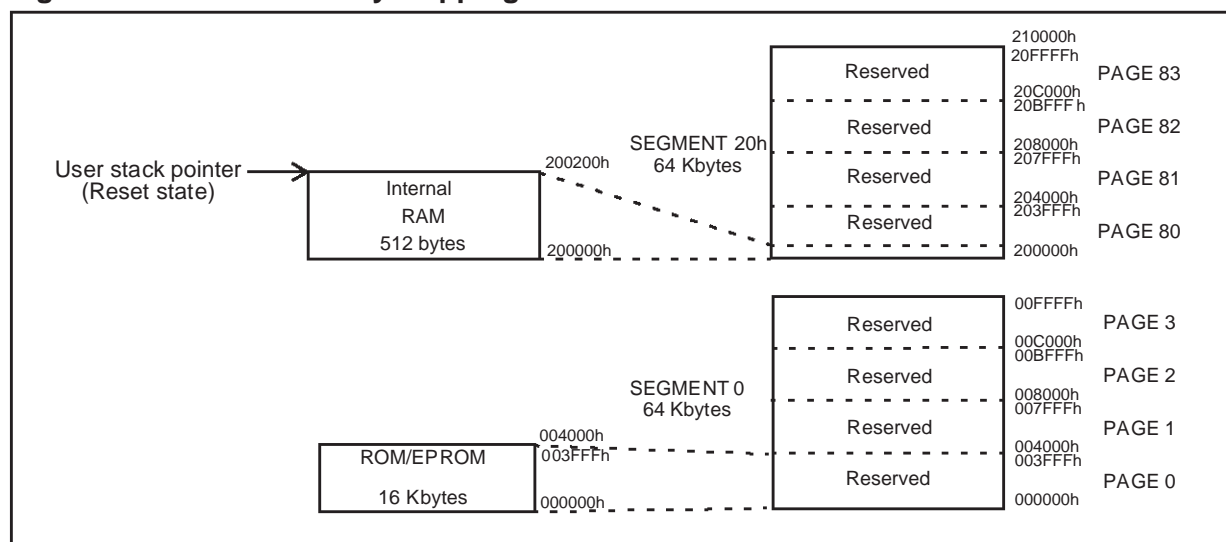
Variables located in RAM are handled by the compiler, but the user must manually manage the register file to use this useful feature.

2.1.1 ACMOTOR.SCR

The acmotor.scr scriptfile is used to map the memory of the modules. This scriptfile redirects the sections defined in the object files at link time (see Figure 4.):

- code (.text section), constants and values of initialized variables in ROM,
- variables and user stack (.data and .bss section) in RAM.

Figure 4. ST92141 Memory Mapping



2.1.2 RAM: Variables & User Stack

The user stack is handled by the C compiler in order to pass parameters through functions, to save context when entering interrupt routines, etc.

The user stack pointer is set to the end of the RAM section in order to obtain the maximum available memory for the stack (for information, the current library needs 35 bytes when using the closed loop demo program). The user must be aware that no tests are performed during compile time at the stack location (i.e. it could overlap the run-time required for areas reserved for variables).

2.1.3 REGISTER FILE

This concerns the C Reserved Area, System Stack and User Free Registers.

2.1.3.1 Mapping

Other than the system registers (R224 to R239) and paged registers (R240 to R255) reserved for peripherals, register files are mapped as shown below:

- The lower part of the register file (Group 0), using the shortest addressing mode (working register mode) is used as the working area for the C-compiler.
- Registers R16 to R99 are reserved for the ST92141 Library.
- The higher part is reserved for the system stack (return addresses, etc.). For reference, 24 bytes are required for library version 1.0 (i.e. from R223 to R199).

Note: This mapping has been modified for Version 2.0. Please refer to application note AN1277 for more details.

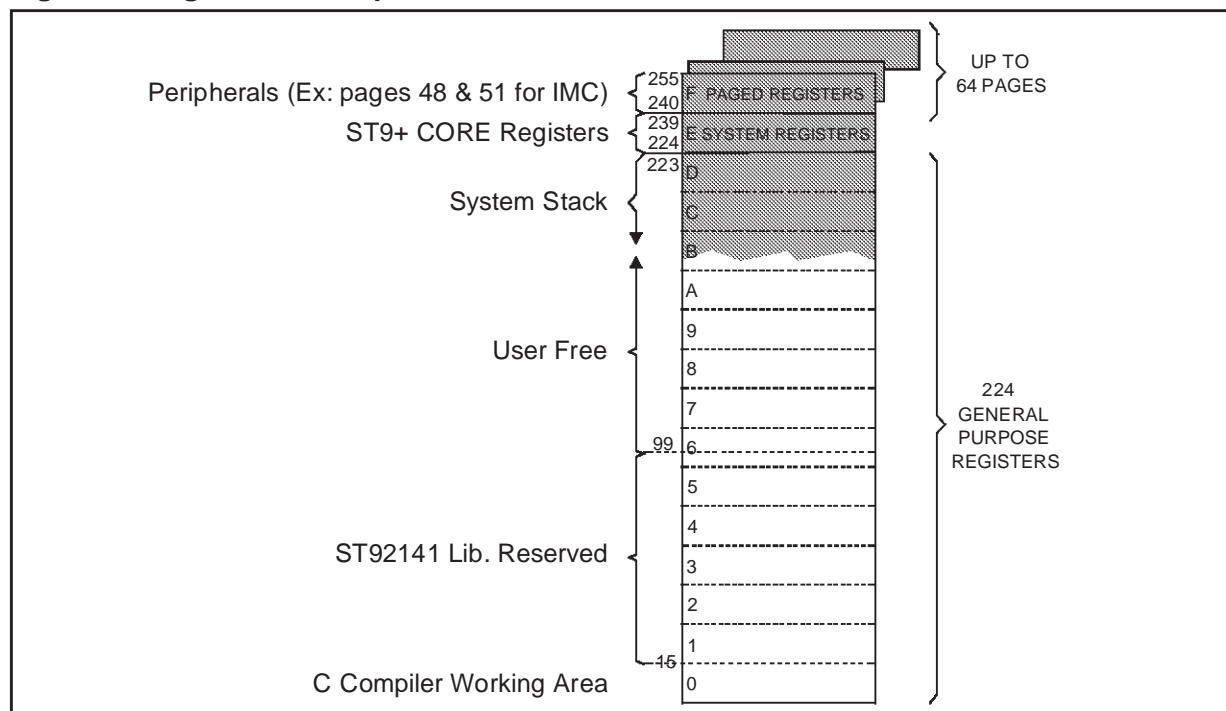
2.1.3.2 Customization

The rest of the register files are user free. Using register file locations as variables provides a higher execution speed and improves code density. In any case, since available free space is limited, registers should only be allocated for the most frequently used variables.

Registers must be declared in the `reg_file.h` header file and obtain a global variable status (i.e. accessible by any module). Some restrictions exist nevertheless:

- A pointer cannot be used for register file variables.
- At compile time, registers that are declared using the `asm` statement cannot be checked for overwriting.

Figure 5. Register File Map



2.2 DEFINE.H

The purpose of this file is to declare objects which will be used throughout the entire library.

- Re-definition of data types using the following convention: a first letter indicating if a variable is signed (s) or unsigned (u), plus a number indicating the number of available bits (example: u8, s16, etc.).
- Defines for assembly mnemonics used in C source code.
- Common macros used to set a page pointer (spp[page]) or to obtain the dimension of an array (DIM[x]).

2.3 CRT9.ASM FILE

The Crt9.asm file is known as the start-up file. It must be linked prior to use to ensure the proper location of resets, NMIs, divide-by-zero traps and other interrupt vectors (imposed by hardware).

It is also used for the following:

- initializing certain system registers (stack pointers, etc.),
- clearing RAM and register files,
- copying the initial values of initialized variables from ROM to RAM.

3 PRESENTATION OF MODULES AND LIBRARY FUNCTIONS

3.1 LIBRARY REFERENCES

Functions are described in the format given below:

Synopsis	This section lists the referenced include files and prototype declarations.
Description	The functions are specifically described with a brief explanation of how they are executed.
Input	In few lines, the format and units are given.
Returns	Gives the value returned by the function, including when an input value is out of range or an error code is returned.
Caution	Indicates the limits of the function or specific requirements that must be taken into account before implementation.
Warning	Indicates important points that must be taken into account to prevent hardware failures.
Functions called	Used to prevent conflicts due to the simultaneous use of resources.
Duration	The approximate duration of the routine with measurement conditions. This is usually performed using the maximum CPU clock periods (25 MHz) without interrupts, unless the duration is a parameter of the function.
Code example	Indicates the proper way to use the function if there are certain prerequisites (interrupt enabled, etc.).

Some of these sections may not be included if not applicable (no parameters, obvious use, etc.).

3.2 IMC MODULE

3.2.1 Description

The “IMCModul.c” module is the part of the hardware layer that is used as an interface between the application layer and the physical world (switches, drivers, speed sensor).

It provides:

- basic setup / control functions for the Induction Motor Controller (IMC),
- PWM interrupt processing,
- speed acquisition related interrupts and functions,
- several functions used to periodically execute various tasks (4 in the current library version) that may need to be asynchronous, in a way that is user-friendly,

The prototype functions are located in the “IMCModul.h” header file.

3.2.2 Interrupts

All available interrupts are declared in this file. Four of the eight interrupts are enabled and described below. The other interrupts must be simply entered.

The ADT_IT and ZPC_IT interrupts are related to the PWM, and the CPT_IT and OTC_IT interrupts are used to handle speed feedback related events. The IMC interrupt priority has been set to 0 (highest priority), if several interrupts occur at the same time, the conflict will be resolved by an internal hardware daisy chain.

3.2.2.1 ADT_IT

An Automatic Data Transfer (ADT) occurs when the values stored in the preload registers are loaded into the active registers.

Note: ADT_IT routines are processed differently in Version 2.0. Please refer to application note AN1277 for more details.

Several tasks are also performed by the software. During this routine, the PWM values that describe the sine wave are loaded from the RAM tables into the IMC peripheral registers. Pointers to the RAM tables are usually increased, unless one of the following events occurs:

- the end of the tables is reached,
- the direction is changed,
- the brake function is enabled,
- one of the active RAM tables is changed (i.e. either the voltage or stator frequency values are changed).

This interrupt occurs at the end of a “step” (several PWM periods having the same duty cycle) when the repetition counter is at zero (a complete sine being described in 48 or 12 steps depending on output frequency). Using preload registers (automatic hardware loading), there

are no real time constraints, but the values loaded will be used for the next step while the current step is already in progress using the values from previous ADT interrupts.

Duration: 12 μ s (average), 25 μ s (maximum, when both the frequency and amplitude of output sine waves are modified).

Also, refer to the corresponding flowcharts in the appendix (Figure 18. and Figure 19. in Section 4.1.1).

3.2.2.2 ZPC_IT

The Zero of PWM Counter (ZPC) interrupt occurs with every PWM cycle. It provides a software timebase by increasing several software timers every ms, with each timer being dedicated to a particular task.

Note: ZPC_IT routines are processed differently in Version 2.0. Please refer to application note AN1277 for more details.

In order to optimize the CPU load, this timebase function can be replaced by other timer resources (the Extended Function Timer or the Standard Timer if not used for the UART) which will only generate one interrupt every 1 ms period.

Duration: Average: 4.5 μ s, Maximum: 10 μ s

Also, refer to Section 3.2.4.3 (IMC Software Watchdog) and the interrupt flowchart that describes the software timebase handling: Figure 19. in Section 4.1.3.

3.2.2.3 CPT_IT

CPT stands for "Capture Tacho counter".

This interrupt is triggered after every active transition of the Tacho input pin. The time interval since the last event is captured in the Tacho Capture register and the Tacho counter are reset (automatically by hardware).

The purpose of this interrupt is first to store this period, which will be used later to compute speed feedback with the correct unit (0.1Hz). The last four values are actually stored in a software stack (FIFO) and will be used later to average the raw results and thus reduce errors due to noise, tachogenerator dissymmetry, etc. The interrupt then performs a "self-setting" routine to maintain accurate speed acquisition throughout the entire speed range.

Duration: Average 7 μ s, Maximum 11 μ s

Also, refer to the corresponding flowchart: Figure 20. in Section 4.1.3.

3.2.2.4 OTC_IT

OTC stands for “Overflow of Tacho Counter”.

This interrupt occurs if the tacho counter rolls over, meaning that either the electrical connection to speed sensor was broken or that the speed has dramatically changed since the previous acquisition (mechanical failure, broken belt, motor in stalled condition, etc.).

A flag is set by this interrupt that is polled by the IMC_GetRotorFreq function. If TRUE, the returned speed will be 0. This flag is systematically reset in CPT_IT.

Useful Tip: In Open Loop control mode (no speed feedback), this interrupt can be used to provide a convenient timebase.

3.2.3 List of Available Functions

The following is a list of available functions as listed in the acmotor.h header file.

IMC_Init (refer to the In-line comments)	
IMC_Start_PWM_Generation	page 20
IMC_Output_PWM_Enabled	page 20
IMC_Output_Fixed_Pattern_Enabled	page 20
IMC_Output_Enabled_for_Datatransfer	page 21
IMC_Output_High_Impedance	page 21
IMC_NMI_Action_Enabled	page 22
IMC_NMI_Action_Disabled	page 22
IMC_Reset_IMCIVR_NMI_Bit	page 23
IMC_SetCompare0Value	page 24
IMC_GetCompare0	page 24
IMC_GetPWMFrequency	page 24
IMC_Brake_Enable	page 25
IMC_Brake_Disable	page 25
IMC_Brake	page 25
IMC_Toggle_Direction	page 26
IMC_Set_ClockWise_Direction	page 25
IMC_Set_CounterClockWise_Direction	page 26
IIMC_Get_WDT_Counter (refer to the introduction to the IMC Module)	
IMC_Clear_WDT_Counter (refer to the introduction to the IMC Module)	
IMC_Load_Reg_LoopTime	page 27
IMC_Load_Sequence_Duration	page 27
IMC_Load_ms_Counter	page 27
IMC_Reg_LoopTime_Elapsed	page 28
IMC_Upload_time_Elapsed	page 28

IMC_Sequence_Elapsed.	page 28
IMC_ms_Counter_Elapsed.	page 28
IMC_Wait_ms	page 29
IMC_InitTachoMeasure.	page 30
IMC_StartTachoFiltering	page 30
IMC_ValidSpeedInfo	page 31
IMC_GetRotorFreq	page 32

Note: Four new functions have been added to cover the functionalities of Version 2.0. Please refer to application note AN1277 for more details.

IMC_Start_PWM_Generation

IMC_Output_PWM_Enabled

IMC_Output_Fixed_Pattern_Enabled

Synopsis

```
#include "IMCModul.h"
void IMC_Start_PWM_Generation (void);
void IMC_Output_PWM_Enabled (void);
void IMC_Output_Fixed_Pattern_Enabled (void);
```

Description

The purpose of these three functions is to modify the IMC peripheral bits dedicated to PWM generation without directly accessing the corresponding registers (hidden variables).

IMC_Start_PWM_Generation

This function starts the PWM counter and therefore the PWM interrupt generation, without modifying the status of the output buffer.

IMC_Output_PWM_Enabled

IMC_Output_Fixed_Pattern_Enabled

These functions act on the IMC peripheral output buffer, enabling either the PWM (out of deadtime generators) for normal operation or a fixed predefined pattern for each of the six PWM outputs. (Inverter off state for example, by imposing a low level on each input of the L6386D level-shifter drivers).

Duration

1.2 μ s for the three routines (CPU running at 25 MHz, without interrupt)

See also

ST92141 Datasheet: IMC chapter.

Code example

```
...
IMC_Init (); /* These 3 first functions are mandatory before
              any call of IMC_Output_XXX_Enabled functions */
CORE_Enable_Interrupts ();
IMC_StartPWM_Generation (); /* Start IMC interrupts generation*/
...
IMC_Output_PWM_Enabled (); /* Supply motor */
...
IMC_Output_Fixed_Pattern_Enabled (); /* Deflux motor windings */
```

IMC_Output_Enabled_for_Datatransfer

IMC_Output_High_Impedance

Synopsis

```
#include "IMCModul.h"
void IMC_Output_Enabled_for_Datatransfer (void);
void IMC_Output_High_Impedance (void);
```

Description

The purpose of these two functions is to modify the IMC peripheral bit acting on the PWM dedicated output ports without directly accessing the corresponding registers (hidden variables).

IMC_Output_Enabled_for_Datatransfer.

This function sets the OPE (Output Port Enable) bit of the OPR register. It allows data (either PWM or a fixed pattern) to be transferred to the port.

IMC_Output_High_Impedance

This function resets the OPE, thus causing the port to be floating (i.e. the logical level may be forced by an external component: pull up/down resistor, level shifter, etc.).

Duration

Maximum: 1.2 μ s (CPU running at 25 MHz, without interrupt)

See also

ST92141 Datasheet: IMC chapter.

IMC_NMI_Action_Enabled IMC_NMI_Action_Disabled

Synopsis	<pre>#include "IMCModul.h" void IMC_NMI_Action_Enabled (void); void IMC_NMI_Action_Disabled (void);</pre>
Description	<p>The purpose of these two functions is to modify the IMC peripheral bit dedicated to NMI functions without directly accessing the corresponding registers (hidden variables).</p> <p>IMC_NMI_Action_Enabled</p> <p>This function sets the NMIE (Non Maskable Interrupt Enable) bit. It allows the NMI input signal to act on the IMC controller, putting PWM ports in high impedance (if the NMIL bit is set accordingly).</p> <p>IMC_NMI_Action_Disabled</p> <p>This function resets the NMIE, thus causing the NMI signal to be directly sent to the core without acting on the IMC controller.</p>
Duration	Maximum: 1.2 μ s (CPU running at 25 MHz, without interrupt).
Note	Neither function will affect the action of the NMI on the ST9 core.
See also	ST92141 Datasheet: IMC chapter.

IMC_Reset_IMCIVR_NMI_Bit

Synopsis	<pre>#include "IMCModul.h" void IMC_Reset_IMCIVR_NMI_Bit (void);</pre>
Description	<p>The purpose of this function is to clear the NMI pending bit of the IM-CIVR register that is dedicated to the IMC peripheral (which is different from the ST9's core NMI flag located in the IVR system register) without directly accessing the corresponding registers (hidden variables).</p> <p>This function also performs a test on NMI bit status immediately after its tentative reset. This is required since this reset will not be effective if the NMI input signal is still active (hardware protection).</p>
Returns	Boolean parameter, TRUE if the NMI bit has been successfully cleared, FALSE otherwise.
Duration	Maximum: 1.9 μ s (CPU running at 25 MHz, without interrupt).
See also	ST92141 Datasheet (IMC section, NMI management chapter).

IMC_SetCompare0Value

IMC_GetCompare0

IMC_GetPWMPFrequency

Synopsis

```
#include "IMCModul.h"

void IMC_SetCompare0Value (void);
void IMC_GetCompare0 (void);
void IMC_GetPWMPFrequency (void);
```

Description

The purpose of these three functions is to read and/or modify the IMC peripheral Compare0 register that is dedicated to PWM generation without directly accessing the corresponding registers (hidden variable).

IMC_GetCompare0

This function is used to read the value of the Compare0 register.

IMC_SetCompare0Value

This function is used to write in the Compare0 register. Since this register has a width of 10-bits, the input parameter (16-bit) will be clamped if its value exceeds 1023.

IMC_GetPWMPFrequency

This function returns the PWM frequency needed to calculate the output signal frequency (see ACM_GetCurrentStatorFreq).

Duration

Values measured with CPU running at 25 MHz, without interrupt.

IMC_GetCompare0: 1.5 μ s.

IMC_SetCompare0Value: 2.4 μ s

IMC_GetPWMPFrequency: 125 μ s

See also

ST92141 Datasheet: IMC chapter.

IMC_Brake_Enable

IMC_Brake_Disable

IMC_Brake

Synopsis

```
#include "IMCModul.h"

void IMC_Brake_Enable (u16 DutyCycle_10bit);
void IMC_Brake_Disable (void);
void IMC_Brake (u16 DutyCycle_10bit, u16 Time_in_ms);
```

Description

IMC_Brake_Enable

This function switches on the active brake of the motor. This is achieved by sinking the DC current in a one-phase winding, the others being maintained to ground. Furthermore, a PWM is systematically switched on to enable the braking function even when the motor is stopped (static braking torque).

IMC_Brake_Disable

This function switches ON/OFF the active braking and returns to normal PWM generation (output signals are continued from the point where they were previously stopped).

IMC_Brake

This function enables the active brake for a given time and then automatically returns to normal PWM generation at the end of this period.

Inputs

Time is given in milli-seconds (ms).

The PWM duty cycle is a u16 variable, but with 10-bit resolution.

Due to the position of the significant bits in the compare registers dedicated to PWM generation in the IMC peripheral, the duty cycle cannot be expressed directly: only b15..b5 will be used. The desired duty cycle must be multiplied by a factor of 32 before being passed to the function.

Example:

Desired duty = 1 / 1024 gives Duty-Cycle_10_bit = 32.

Desired duty = 100 / 1024 gives Duty-Cycle_10_bit = 3200.

Desired duty = 100% gives Duty-Cycle_10_bit = 32768.

Duration

IMC_Brake_Enable: 167 μ s (CPU running at 25 MHz, without interrupt)

IMC_Brake_Disable: 2.2 μ s (CPU running at 25 MHz, without interrupt)

Warning

Since limitations are not implemented on the PWM duty cycle range, the DC current may grow very quickly when this duty is increasing, depending on the motor winding characteristics.

IMC_Set_ClockWise_Direction

IMC_Set_CounterClockWise_Direction IMC_Toggle_Direction

Synopsis	<pre>#include "IMCModul.h" void IMC_Set_ClockWise_Direction (); void IMC_Set_CounterClockWise_Direction (); void IMC_Toggle_Direction ();</pre>
Description	<p>These functions are used to set and/or modify the rotating field direction. This is achieved by swapping 2 phase pointers.</p> <p>The clockwise direction is set randomly. The real direction will only depend on the physical connection of the motor.</p>
Duration	1.1 μ s for the three routines (CPU running at 25 MHz, without interrupt)
Warning	<p>No tests are performed on motor status (running or stopped) inside these functions.</p> <p>The user must therefore be sure that motor is stopped before calling any of these three routines, otherwise the operation quadrant could suddenly be changed to generator, thus sinking the reactive current in the high voltage DC bus and re-charging the bulk capacitor over its maximum voltage rating.</p>

IMC_Load_Reg_LoopTime

IMC_Load_ms_Counter

IMC_Load_Sequence_Duration

Synopsis	<pre>#include "IMCModul.h" void IMC_Load_Reg_LoopTime (u8 ms); void IMC_Load_Sequence_Duration (u16 ms); void IMC_Load_ms_Counter (u16 ms);</pre>
Description	<p>These functions are used to setup the duration or intervals of the related processes:</p> <ul style="list-style-type: none">– Regulation Loop time,– Frequency modification along ramps (sequence of steps)– User free process (ms_counter) <p>These functions must be called before the following functions are used: IMC_Reg_LoopTime_Elapsed, IMC_Sequence_Elapsed or IMC_ms_Counter_Elapsed.</p>
Inputs	<p>The inputs are the desired duration / time intervals.</p> <p>Note: The time base is 1 ms. This unit can be modified in the imc.c define section.</p>
Duration	1.5 μ s (CPU running at 25 MHz, without interrupt)
See also	IMC_Reg_LoopTime_Elapsed, IMC_Sequence_Elapsed, MC_ms_Counter_Elapsed in section 3.2.4.2 on page 34

IMC_Reg_LoopTime_Elapsed

IMC_Upload_time_Elapsed

IMC_Sequence_Elapsed

IMC_ms_Counter_Elapsed

Synopsis	<pre>#include "IMCModul.h" BOOL IMC_Reg_LoopTime_Elapsed (void); BOOL IMC_Upload_time_Elapsed (void); BOOL IMC_Sequence_Elapsed (void); BOOL IMC_ms_Counter_Elapsed (void);</pre>
Description	<p>These functions are used for polling if one of the related periods / durations has elapsed:</p> <ul style="list-style-type: none">- Regulation loop time,- Upload period for data transmission to a terminal (PC, etc.) (The upload timebase value cannot be modified during the runtime (constant) since it is usually terminal-dependant. The value is set to 10ms, in IMCParam.h (_UploadPeriod)),- Frequency step before frequency increase/decrease,- User free sequence.
Caution	<p>Before calling the above functions, the user must have properly setup the duration with the following functions: IMC_Load_Reg_LoopTime (), IMC_Load_Sequence_Duration (), IMC_Load_ms_Counter ().</p>
Returns	<p>Boolean variable, TRUE <i>only the first time the function is called</i> after programmed duration is elapsed, FALSE otherwise.</p>
Duration	<p>1.5 μs (CPU running at 25 MHz, without interrupt)</p>
Code example	<pre>IMC_Init (); /* These 3 first functions are mandatory before any call of timing functions */ CORE_Enable_Interrupts (); IMC_StartPWM_Generation (); ... IMC_Load_ms_Counter (TimeOut); while (! (IMC_ms_Counter_Elapsed ())) { DoTheJob(); /* Until TimeOut elapsed */ }</pre>
See also	<p>section 3.2.4.2 on page 34</p>

IMC_Wait_ms

Synopsis	<pre>#include "IMCModul.h" void IMC_Wait_ms (u16 Time_in_ms);</pre>
Description	This function provides a basic timing function. When entered, it returns only once the Time Duration has elapsed and does not perform any actions during this period.
Inputs	The input is the duration of the routine in ms.
Functions called	IMC_Load_ms_Counter IMC_ms_Counter_Elapsed
Caution	This function uses the ms_Counter variable and the 2 above functions for timing. These resources must therefore not be used when calling the IMC_Wait_ms function.

IMC_InitTachoMeasure IMC_StartTachoFiltering

Synopsis

```
#include "IMCModul.h"
void IMC_InitTachoMeasure ();
void IMC_StartTachoFiltering ();
```

Description

IMC_InitTachoMeasure

The purpose of this function is to initialize the flags and variables associated with speed acquisition: the SW stack where the last 4 speed acquisitions are stored, the Tacho clock prescaler for the low speed range and the flag disabling rolling average. The IMC_GetRotorFreq function will return a speed calculated from the last acquisition.

IMC_StartTachoFiltering

Once called, this function enables the IMC_GetRotorFreq to return a speed corresponding to the average of the last values.

Duration

Values measured with CPU running at 25 MHz, without interrupt.

IMC_InitTachoMeasure: 3.6 μ s

IMC_StartTachoFiltering: 1.3 μ s

Code example

```
...
...
IMC_InitTachoMeasure(); /*Must be called when motor is stopped*/
...
if (IMC_ValidSpeedInfo (MinRotorFreq))
{
    IMC_StartTachoFiltering (); /* Must be called once we
    are sure that we have reliable speed information */
}
```

See also

IMC_GetRotorFreq, IMC_ValidSpeedInfo.

IMC_ValidSpeedInfo

Synopsis	<pre>#include "IMCModul.h" BOOL IMC_ValidSpeedInfo (u16 MinRotorPeriod);</pre>
Description	<p>The purpose of this function is to know if the rotor shaft turns at the correct speed.</p> <p>This means:</p> <ul style="list-style-type: none">- there were at least 3 speed values (i.e. 3 active edges on the Tacho input pin),- the acceleration is positive (i.e. time between edges is decreasing),- Rotor speed has reached the desired value to enable further operation: regulation, ramp-up, etc. The rotor speed has entered the intrinsically stable tile of the torque versus frequency characteristic. <p>Note: Acceleration detection is used to distinguish reliable values from the very first ones when the time between tacho edges was greater than the Tacho counter roll-over period. Acceleration sensitivity may be adjusted in the IMCParam.h file (DeltaMin value): the higher the value, the higher the acceleration detection threshold.</p>
Input	The input parameter is the minimum rotor speed at which the motor is considered as really being started.
Returns	Boolean parameter, TRUE if the three above conditions are verified, unless a function has been called with the MinRotorPeriod equal to 0. This is known as Open Loop mode and will disable the tests listed above and cause the function to return FALSE.
Duration	Maximum: 104 μ s (CPU running at 25 MHz, without interrupt)
Caution	<ol style="list-style-type: none">1. There is no way to differentiate rotation directions using a tachogenerator. The user must be aware that this routine may return TRUE in certain conditions, even if the motor is not started in the right direction. In this case, the user should manage a minimum amount of time before re-starting (i.e. high inertia load). Obviously, this function may be ineffective if the start-up duration is far shorter than time needed to have 3 consecutive speed values.2. Be careful when setting the DeltaMin value in the IMCParam.h file. This value is given using the tacho timer time unit (i.e. depending on the prescaler). Refer to Section 3.2.4.5.

IMC_GetRotorFreq

Synopsis	<pre>#include "IMCModul.h" u16 IMC_GetRotorFreq (void);</pre>
Description	<p>The purpose of this function is to provide the rotor frequency with an accuracy that is equivalent to the sine wave generator function.</p> <p>The frequency is calculated using two parameters: the acquired speed sensor signal period and the tacho counter prescaler.</p> <p>Depending on previous calls to the IMC_InitTachoMeasure and IMC_StartTachoFiltering functions, the acquired period could be used raw or as an average of the last 4 periods.</p>
Returns	<p>Rotor frequency with [0.1Hz] unit.</p> <p>If the calculated speed is less than the minimum measurable speed or if a tacho counter overflow has occurred, the returned value will be 0.</p> <p>Note: The user must properly handle the case when a tacho overflow occurs inside the OTC_IT interrupt (stop the motor, re-initialize tacho acquisition, etc.). In any case, the user can rely on the 0 value returned by this function to detect any speed feedback error conditions, as the tacho overflow flag is cleared each time the Tacho capture interrupt is triggered.</p>
Duration	Maximum: 110 μ s (CPU running at 25 MHz, without interrupt)
Caution	<p>The rolling average is effective only if the number of tacho captures between two IMC_GetRotorFreq calls is less than or equal to one.</p> <p>At high speed, the tacho capture period is usually much lower than the period between two calls of the IMC_GetRotorFreq. In this case, the rolling average becomes a standard average of the last four acquired speed values.</p>
See also	<p>Customization issues in Section 3.2.4.4. and Section 3.3.3.5.</p> <p>Flowchart in Section 4.1.4: Figure 21.</p>

3.2.4 Detailed Explanations and Customization

3.2.4.1 ST92141 Library Main Timebase

Most of the control task periods or durations are setup using the software timebases previously described (see also Section 3.2.4.2).

The basic unit for this time setting is 1 ms. It is achieved using the ZPC_IT resource (see Section 3.2.2.2). Since this interrupt occurs at every PWM cycle (82 μ s period), 12 interrupts will correspond roughly to 1 ms (982 μ s). The ZPC_IT interrupts are counted in a PWM Tick-Counter variable up to the TIMEBASE constant value (currently set to 12) before modifying the various software timers, etc. (see Section 3.2.2.2).

The constant TIMEBASE value is set in the IMCParam.h define section.

Note: Since this 1 ms timebase is coupled with the PWM frequency, if a PWM frequency change occurs, the SW timebase will be modified accordingly.

The current sine wave generation slightly modifies the PWM frequency at high speeds. The worst case for timebase errors is:

- 5% for a maximum stator frequency of 339.4 Hz,
- 10% for a maximum stator frequency of 680 Hz.

Note: This remark no longer applies to Version 2.0 sine wave generation. In this case, the timebase error remains constant at 1.8% (982 μ s, instead of 1 ms).

If a high precision timebase is required for the application (i.e. better than 10%) or if it must work without a running IMC peripheral, other timer resources must be considered (extended function timer, standard timer if not used for UART).

3.2.4.2 Software Timebase

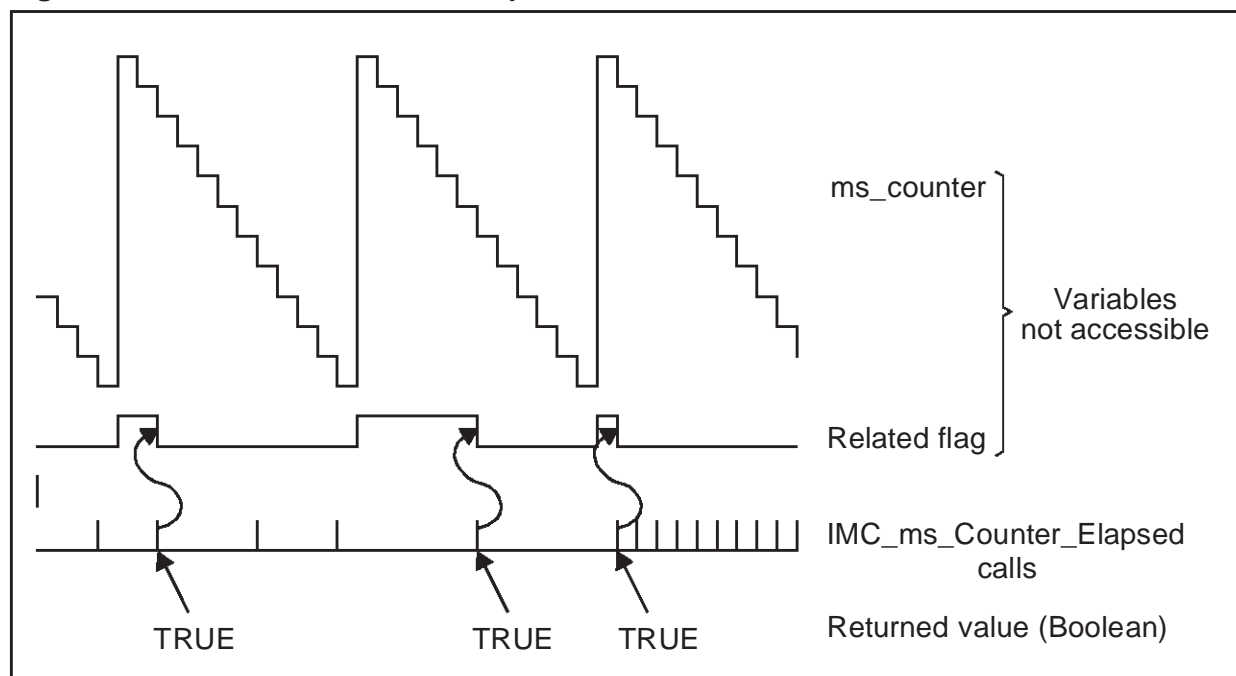
Several software timebases have been set for the periodical and asynchronous sequencing of various tasks (regulation loop time, serial link data exchanges rate, etc.) which may have different periods.

These timebases were designed to be used in Polling mode and take advantage of the ZPC_IT interrupts issued by the IMC cell. Obviously, time data will be lost if:

- the IMC_XXXX_Elapsed functions are not polled frequently enough,
- the timebase duration was not entered (functions IMC_Load_XXXX) correctly.

Figure 6., shown below, presents how the IMC_ms_Counter_Elapsed function works.

Figure 6. Software Timebase Principle



From this figure, it is clear that IMC_ms_Counter_Elapsed will return TRUE if the predefined duration has been completed, but without any information on the time elapsed since this event occurred. A while structure will therefore minimize jittering:

```
while (!(IMC_ms_Counter_Elapsed ()))
{
    DoTheJob();          /* while duration is not elapsed */
}
```

3.2.4.3 IMC Software Watchdog

The IMC_WDT_Counter variable is increased every ms (up to 255, without roll-over) in the ZPC_IT routine.

The purpose is to monitor, in the main program, the number of interrupts that have been issued between 2 main loop scan periods (assuming that this period is roughly constant and known).

If this number exceeds a predefined range, an error has occurred (i.e. the PWM has been stopped, external interrupts have overloaded the CPU, etc.). The user is then free either to correct the error or to let the hardware watchdog reset the MCU.

This IMC_WDT_Counter variable can only be accessed by function calls to have a Read-only like variable which cannot be accidentally modified by other modules.

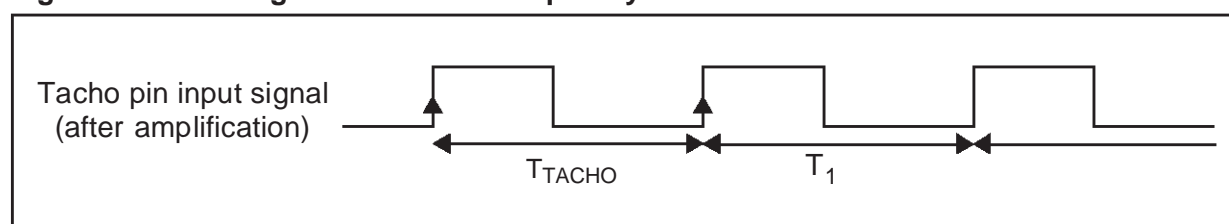
Further explanations may be found either in the routine documentation in the software library or with the module's in-line comments.

3.2.4.4 IMC_GetRotorFreq

In order to obtain speed feedback, the IMC peripheral contains a dedicated timer. The method used to determine the rotor frequency is shown in Figure 7. (*Tacho* refers to the tacho-generator, an usually low-cost speed sensor). Several parameters must be taken into account:

- The number of pulses per rotor revolution will depend on the sensor (either the position sensor: hall, etc. or the velocity sensor: tacho generator with n number of poles, etc.).
- To obtain the required accuracy (0.1 Hz) throughout the entire speed range, the dynamic range of a 16-bit capture register is not enough. The tacho counter input clock must be prescaled according to the rotor frequency that is to be measured.

Figure 7. Tacho Signal and Rotor Frequency Calculation



$$T_{TACHO} = (TPRS + 1) \times T_{CPU} \times \text{Capture gives } F_{TACHO} = F_{CPU} / ((TPRS + 1) \times \text{Capture})$$

Finally, the following is calculated:

$$F_{ROTOR[0.1\text{ Hz}]} = (10 \times F_{CPU[Hz]}) / (N \times (TPRS + 1) \times \text{Capture})$$

Where:

$TPRS$ is the Tacho Prescaler value

Capture is the Tacho counter captured value

F_{CPU} is the CPU clock frequency

N is the number of pulses per rotor revolution.

Note: (TPRS + 1) represents the real division factor. When TPRS = 0, there is no prescaling.

The TPRS value will change during the run-time in CPT_IT, assuming the following:

- IMC_GetRotorFreq must be called after IMC_InitTachoMeasure when the motor is stopped, in order to make sure that TPRS value is pre-set for the low speed domain,
- There will not be any sudden frequency changes implying changing the TPRS from the minimum to the maximum value (see Figure 8.).

Figure 8. TPRS Setting according to Rotor Frequency

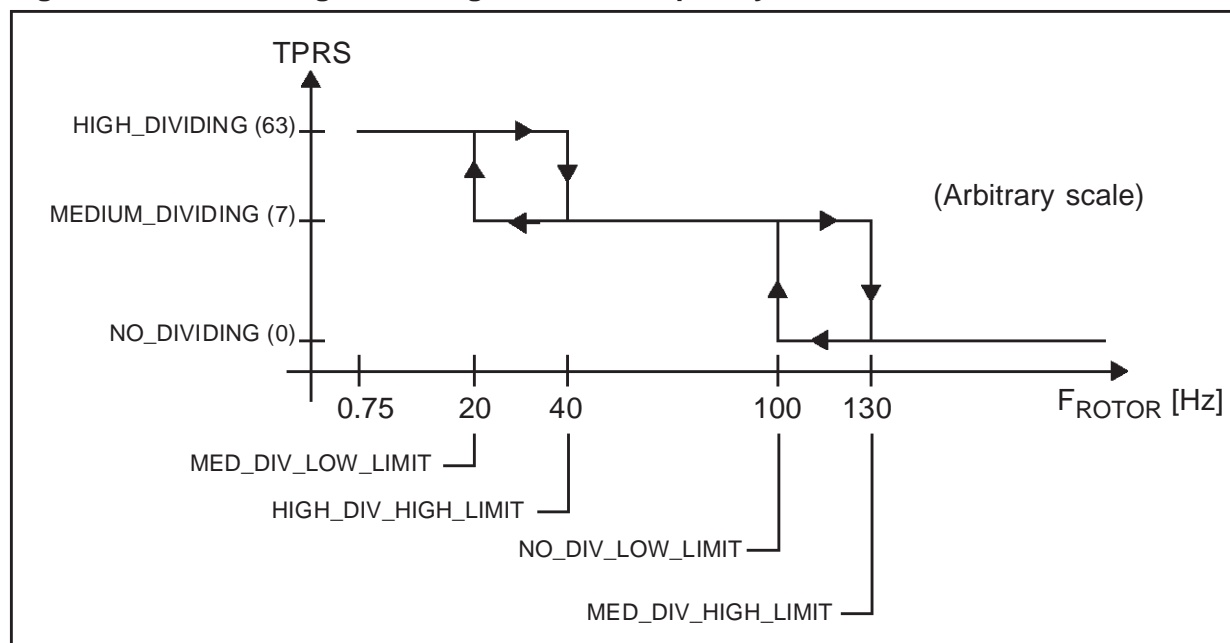


Table 1. Tacho Counter Captured Values (Accuracy given for worst-case only)

F_{ROTOR} (Hz)	20	40	100	130	670
Prescaling Clock / 64	2440	1220 (± 0.01 Hz)			
Prescaling Clock / 8	19530	9765	3906	3004 (± 0.02 Hz)	
Prescaling Clock / None			31251	24038	4644 (± 0.07 Hz)

The accuracy can be easily calculated using the equations given below. For example: accuracy when rotor frequency is 670 Hz (i.e. a speed of 20,000 RPM with a 4-pole motor and a 16-pole tachometer giving 8 pulses / turn)

$$Capture = \frac{31250000}{6700} \cong 4644$$

4644 gives $F_{\text{ROTOR}[0.1 \text{ Hz}]} \cong 6729.11$

4643 gives $F_{\text{ROTOR}[0.1 \text{ Hz}]} \cong 6730.56$ (i.e. a delta of 1.45 giving $\pm 0.07 \text{ Hz}$)

Assuming that:

$N = 8$

$TPRS = 0$

$CPU \text{ Clock} = 25 \text{ MHz}$

3.2.4.5 Customizing Rotor Frequency Acquisition

Depending on the system parameters (sensor characteristics, etc.), the user must modify the following defines in the IMCParam.h header file:

```
#define NO_DIVIDING ((u16)0) /* WARNING: this values are cast to */
#define MEDIUM_DIVIDING ((u16)7) /* u8 in CPT interrupt */
#define HIGH_DIVIDING ((u16)63)

#define HIGH_DIV_HIGH_LIMIT ((u16)1220)
#define MED_DIV_LOW_LIMIT ((u16)19530)
#define MED_DIV_HIGH_LIMIT ((u16)3004)
#define NO_DIV_LOW_LIMIT ((u16)31251)

#define PULSES_PER_TURN ((u8)8)
```

The basic idea is to adapt dividing ratios and domain limits to maintain a sufficient accuracy. Certain points must be carefully verified:

- dividing ratios are cast in CPT_IT; if greater than 255, the corresponding code and associated variable “PreviousTPRS” must be modified.
- if capture limits (HIGH_DIV_HIGH_LIMIT, MED_DIV_LOW_LIMIT, . . .) are not properly setup, Tacho counter overflows may occur, causing IMC_GetRotorFreq to return 0 (see Section 3.2.4.6)
- PULSES_PER_TURN is the number of logical pulses issued (directly or after amplification) by the speed sensor after each mechanical revolution of the rotor.

See also flowchart in Section 4.1.4 (Figure 21.)

3.2.4.6 Tacho Compare Event Issues

The IMC cell contains a compare register dedicated to abnormal speed signal detection. Since the comparison is only performed on the highest bits of the tacho counter, the maximum value that can be captured cannot be 0xFFFF.

As an example, if the compare register is set to FF, the related OTC interrupt will be issued when the tacho counter reaches 0xFF00 and the maximum period that can be captured without the OTC_IT interrupt will be 0xFEFF. The corresponding minimum frequency which can be measured will then depend on the Tacho counter clock prescaler (see Table 2).

Table 2. Minimum Measurable Frequency

TPRS	63	7	0
F _{ROTOR} (Hz)	0.75	6	50

Note: Theoretical values, except for TPRS = 63.

3.3 ACMOTOR MODULE

3.3.1 Description

The purpose of this module is to provide the basic drivers used for generating 3-phase sine waves and controlling AC motors. The selected control method is scalar which is more suitable for low-cost drive systems.

Among the set of functions described below, the following can be distinguished:

- ACM_SoftStart, ACM_SoftStart_OpenLoop, ACM_RampUp, ACM_SustainSpeed are used here as an example, their structure may be modified to implement a state machine in the main program (for example, to avoid spending the complete speed ramp duration inside the related routine and to return periodically in the main routine to refresh the watchdog).
- ACM_StopInverter, ACM_Update_Sine_Tables, ACM_VoltageMaxAllowed, ACM_GetCurrentStatorFreq, ACM_GetCurrentVoltage, ACM_GetCurrentSlip, ACM_InitSlipFreqReg, ACM_SlipRegulation, ACM_GetOptimumSlip, which are ready-to-use functions (duration is already set to minimum).

The rest are listed here for information and are used to facilitate in-depth customization.

Note: Two assembly routines are also declared in the acmotor.h header file¹. This makes them available for C functions. See "sine.asm" in-line comments for more information.

3.3.2 List of Available Functions

As listed in the acmotor.h header file.

ACM_Init (refer to in-line comments)

ACM_SoftStart	page 40
ACM_SoftStart_OpenLoop	page 41
ACM_RampUp	page 42
ACM_SustainSpeed	page 43
ACM_StopInverter.	page 44
ACM_Update_Sine_Tables ²	page 45
ACM_FreqToPeriod ¹	page 46
ACM_SetNewPWMFreq ¹	page 47
ACM_VoltageMaxAllowed.	page 48
ACM_GetCurrentStatorFreq	page 49
ACM_GetCurrentVoltage	page 49
ACM_GetCurrentSlip.	page 49
ACM_InitSlipFreqReg	page 50
ACM_SlipRegulation	page 51
ACM_GetOptimumSlip	page 52
ACM_GetFuzzyError.	page 53
ACM_SmoothSlip32	page 54
ACM_SmoothTrend4.	page 55

Note 1: These routines are obsolete when using Version 2.0. Please refer to application note AN1277 for more details.

Note 2: The prototype of this function has changed in Version 2.0. Please refer to application note AN1277 for more details.

ACM_SoftStart

Synopsis

#include "acmotor.h"

BOOL ACM_SoftStart (u16 StatorFreq, u16 MinRotorFreq);

Description

This function provides the soft start used to limit the inrush current in the motor, while monitoring the speed feedback in order to stop the voltage increase when the starting torque is reached.

Open loop (i.e. MinRotorFreq = 0)

The function increases the stator voltage linearly up to its maximum value (giving the maximum available torque / maximum current in the windings as defined in ACM_VoltageMaxAllowed function), with the stator frequency remaining constant, during SOFTSTART_TIMEOUT (constant given in ms).

Closed loop (i.e. MinRotorFreq != 0)

The function increases the stator voltage linearly, with the stator frequency remaining constant.

The ramp ends when one of the following conditions occurs:

- the SOFTSTART_TIMEOUT is elapsed,
- the motor starts and the rotor speed reaches MinRotorFreq.

At the end of the ramp, if the rotor speed is too low, the tachometer is monitored for an additional set time (user defined OPTIONAL_TIMEOUT constant) to take into account large inertia loads. At the end of this time period, if the rotor speed is still not high enough, the motor is stopped and the function returns FALSE.

Helpful Tip: The comparison between the maximum available voltage and the current voltage at the end of ACM_SoftStart can be used as an indicator of the motor load.

Inputs

StatorFreq is given with [0.1Hz] unit.

MinRotorFreq is given with [0.1Hz] unit.

Returns

Boolean: always TRUE in open loop, otherwise based on speed sensor feedback.

The function returns FALSE if MinRotorFreq is higher than StatorFreq or if the soft start procedure has failed.

Functions called

ACM_VoltageMaxAllowed, ACM_InitTachoMeasure, ACM_Update_Sine_Tables, IMC_Output_PWM_Enabled, IMC_Load_Sequence_Duration, IMC_Sequence_Elapsed, IMC_ValidSpeedInfo, IMC_StartTachoFiltering, ACM_StopMotor.

See also

See "IMC_ValidSpeedInfo" on page 31., Customization issues in Section 3.3.3.4.

Flowchart: Figure 22. in Section 4.2.1.

ACM_SoftStart_OpenLoop

Synopsis	<pre>#include "acmotor.h" void ACM_SoftStart_OpenLoop(u16 StatorFrequency, u16 Duration);</pre>
Description	<p>This function provides a Soft Start used to limit the starting torque and the inrush current in the motor. The voltage on the stator windings is increased until it reaches its maximum allowed value (set in the <code>ACM_VoltageMaxAllowed</code> function).</p> <p>The duration of the Soft Start can be adjusted; its acceleration rate will depend on voltage value returned by the <code>ACM_VoltageMaxAllowed</code> function.</p> <p>This function is more compact than the <code>ACM_SoftStart</code> function used for software that does not use sensors to obtain feedback speed data.</p>
Inputs	<p>StatorFreq is given with [0.1Hz] unit.</p> <p>Duration in ms.</p>
Functions called	<p><code>ACM_VoltageMaxAllowed</code>, <code>ACM_Update_Sine_Tables</code>, <code>IMC_Output_PWM_Enabled</code>, <code>IMC_Load_Sequence_Duration</code>, <code>IMC_Sequence_Elapsed</code>.</p>

ACM_RampUp

Synopsis	<pre>#include"acmotor.h" void ACM_RampUp (u16 EndFrequency, u16 RampTime);</pre>
Description	<p>The purpose of this function is to increase the speed of the motor, from its current speed to the desired one, within the predefined time limit.</p> <p>This speed ramp is achieved using a closed loop slip control. It means that the stator frequency is increased by the acceleration rate calculated from the two inputs, while the voltage is controlled by the regulator to maintain a constant slip during the ramp. The rotor speed will therefore follow the stator frequency ramp with a constant offset that corresponds to the optimum slip. To avoid breakdown conditions, the slip is monitored during the ramp. If a predefined limit is exceeded (SLIP_LIMIT constant, set in acmparam.h header file), the stator frequency will be maintained until a given value in the allowed range is recovered (i.e. from 0 to SLIP_LIMIT).</p> <p>This may occur in the following cases:</p> <ul style="list-style-type: none">– The desired acceleration exceeds the gain / bandwidth capabilities of the regulation algorithm,– The torque exceeds motor ratings. <p>In such conditions, the priority is given to speed rather than time: End-Frequency will finally be reached (if slip recovers its nominal value), but after the programmed time.</p>
Inputs	<p>EndFrequency is the final stator frequency given with [0.1Hz] unit.</p> <p>RampTime is given in ms.</p>
Functions called	<p>ACM_GetCurrentStatorFreq, IMC_Load_Sequence_Duration, IMC_Sequence_Elapsed, ACM_GetCurrentSlip, ACM_GetOptimumSlip, ACM_Update_Sine_Tables, IMC_Reg_LoopTime_Elapsed, ACM_SlipRegulation.</p>
Caution	<p>Before calling this function, the user must have properly setup the regulation loop time in the IMC_Load_Reg_LoopTime function.</p>
See also	<p>Flowchart (Figure 23. in Section 4.2.2).</p>

ACM_SustainSpeed

Synopsis	<pre>#include "acmotor.h" void ACM_SustainSpeed(u16 Time);</pre>
Description	This function simply maintains the current speed on the motor for a desired time. This is achieved using a closed loop slip control that maintains the optimum voltage level in steady state conditions.
Inputs	Time is given in ms.
Functions called	IMC_Load_ms_Counter IMC_ms_Counter_Elapsed IMC_Reg_LoopTime_Elapsed ACM_GetCurrentStatorFreq ACM_GetOptimumSlip ACM_SlipRegulation ACM_Update_Sine_Tables.
Caution	Before calling this function, the user must have properly setup the regulation loop time in the IMC_Load_Reg_LoopTime function.

ACM_StopInverter

Synopsis	<pre>#include "acmotor.h" void ACM_StopInverter (void);</pre>
Description	The purpose of this function is to switch off all power switches and to de-gauss stator windings. This causes the motor to "coast".
Inputs	Time is given in ms.
Duration	2 μ s (CPU running at 25 MHz, without interrupt).
Functions called	IMC_Output_Fixed_Pattern_Enabled
Caution	<p>The inverter switch-off state must have been properly entered in the dedicated IMC OPR register (R252, page 48).</p> <p>Stopping the inverter does not mean that motor will stop immediately. The user must make sure that speed really reaches zero before attempting to restart the motor, especially when driving a very inertial load.</p>

ACM_Update_Sine_Tables

Synopsis	<pre>#include "acmotor.h" BOOL ACM_Update_Sine_Tables (u8 NewVoltage, u16 NewStatorFrequency);</pre>
Description	<p>This function is called each time a new 3-phase sine wave is required (i.e. if either the voltage or output frequency must be changed). It is used to set flags and to call related routines for updating sine wave tables.</p> <p>This routine will limit the frequency within the range defined in the constants section of the acmotor.h file (i.e. between the LOWEST_FREQUENCY and the HIGHEST_FREQUENCY frequencies).</p> <p>Then, the Calc_sin and Calc_rpt_tab assembly routines are called to fill in the RAM tables containing the sine wave coefficients (PWM duty cycles and step lengths).</p>
Inputs	<p>NewStatorFrequency is given with [0.1Hz] unit.</p> <p>NewVoltage is the value of the modulation index, in 8-bit format. The 0 to 100% modulation index corresponds to the 0 to 255 range.</p>
Returns	<p>Boolean type variable:</p> <p>FALSE, if RAM tables are not yet updated after last call of the UpdateSineTables. This is required for very low speeds where the duration of every step (and therefore the time needed to refresh PWM pointers in ADT_IT) may be longer in comparison to other tasks. For example, if the output sine is 5 Hz, every step lasts about $200\text{ ms}/48 = 4\text{ ms}$.</p>
Duration	890 μs (CPU running at 25 MHz, without interrupt)
Functions called	<ul style="list-style-type: none">– Calc_sin (assembly routine),– Calc_rpt_tab (assembly routine),– ACM_GetCurrentStatorFreq– ACM_FreqToPeriod– ACM_SetNewPWMFreq
Caution	<p>No tests are performed in this function on the input parameters, except for the frequency range. The following conditions must therefore be verified by the user before making any calls:</p> <ul style="list-style-type: none">– compliance with the maximum voltage and frequency characteristics for the motor,– correct torque or current transients (i.e. their values do not vary too quickly when motor is running).
See also	Flowchart (Figure 24. in Section 4.2.3).

Note: The prototype of this function has changed in Version 2.0. Please refer to application note AN1277 for more details.

ACM_FreqToPeriod

Synopsis	<pre>#include "acmotor.h" u16 ACM_FreqToPeriod(u16 StatorFrequency);</pre>
Description	<p>The purpose of this function is to determine the most suitable "period" for a given sine wave output frequency. This variable contains the total number of PWM cycles needed to describe a sine wave period; this number is therefore an integer, given without a time unit.</p> <p>The method used to calculate the "period" value will depend on the output frequency range and the PWM operating frequency.</p> <p>At low frequency, PWM switching is kept constant.</p> <p>At high speed, it will be variable to maintain a good resolution of the stator frequency and must be paired to balance both half-waves of the sine wave and to prevent any parasitic DC current injections.</p>
Inputs	StatorFrequency is given with [0.1Hz] unit.
Returns	Period value as a number of PWM cycles.
Duration	117 μ s (CPU running at 25 MHz, without interrupt).
See also	ACM_SetNewPWMFreq.

Note: This routine is obsolete when using Version 2.0. Please refer to application note AN1277 for more details.

ACM_SetNewPWMFreq

Synopsis	<pre>#include "acmotor.h" void ACM_SetNewPWMFreq (u16 NewStatorFrequency, u16 NewPe- riod);</pre>
Description	This function modifies the IMC compare register that defines the PWM frequency based on the new stator frequency and new Period settings. Adjusting PWM frequency slightly modifies the frequency of the output sine wave signal.
Inputs	NewStatorFrequency is given with [0.1Hz] unit. NewPeriod is the number of PWM cycles in a sine wave period.
Duration	Maximum: 140 μ s (CPU running at 25 MHz, without interrupt).
Functions called	IMC_GetPWMFrequency IMC_SetCompare0Value
Caution	<p>This function must be called with the period returned from ACM_FreqToPeriod.</p> <p>The functions must be called in the following order:</p> <ol style="list-style-type: none">1. ACM_FreqToPeriod,2. Calc_sin (assembly routine),3. ACM_SetNewPWMFreq.

Note: This routine is obsolete when using Version 2.0. Please refer to application note AN1277 for more details.

ACM_VoltageMaxAllowed

Synopsis

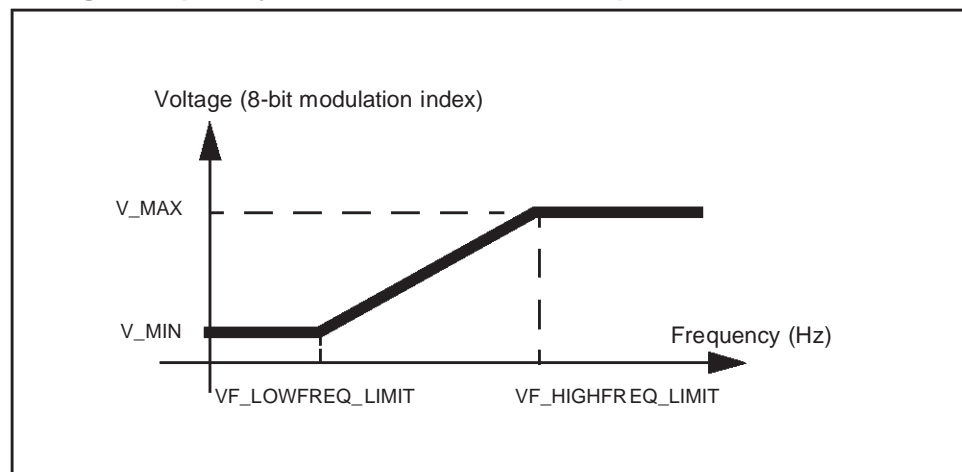
```
#include "acmotor.h"
```

```
u8 ACM_VoltageMaxAllowed (u16 StatorFrequency);
```

Description

This function is used to maintain a constant voltage versus a frequency ratio throughout the entire speed range. The characteristic points of the corresponding curve are represented in the figure below and can be set in the acmotor.h file. (See customization chapter of ACMOTOR module for cautions.)

Figure 9. Standard Voltage/Frequency Characteristics and Setpoints



This is considered as the maximum value since the voltage applied to static windings can be decreased when running a load that is below the maximum motor torque, in order to obtain the required torque and to limit ohmic losses, in Closed Loop operation (i.e. below the rated flux operation).

Therefore, the voltage vs frequency curve must be set for the maximum acceptable current (given by the motor manufacturer) in Closed Loop mode or set for nominal current in Open Loop mode.

Inputs

StatorFrequency is given with [0.1Hz] unit.

Duration

Maximum: 7.5 μ s (CPU running at 25 MHz, without interrupt)

Returns

Modulation index voltage (u8 variable).

If the Frequency parameter is outside the available domain, the returned value will be VOLTAGE_MIN / VOLTAGE_MAX (no error code returned).

ACM_GetCurrentStatorFreq

ACM_GetCurrentVoltage

ACM_GetCurrentSlip

Synopsis	<pre>#include "acmotor.h" u16 ACM_GetCurrentStatorFreq(void); u16 ACM_GetCurrentSlip(void); u8 ACM_GetCurrentVoltage(void);</pre>
Description	<p><i>ACM_GetCurrentStatorFreq</i></p> <p>This function returns the current Stator frequency, calculated from the current number of PWM cycles during the ramp (i.e. the Period variable) and the PWM switching frequency.</p> <p><i>ACM_GetCurrentVoltage</i></p> <p>This function returns the current modulation index.</p> <p><i>ACM_GetCurrentSlip</i></p> <p>This function returns the difference between the stator and rotor frequencies. This value will always be positive (unsigned variable) assuming that negative slip operations (i.e. motor used as a generator) are not designed for this software library. Nonetheless, if the slip is negative, the returned value will be zero.</p>
Returns	Stator and slip frequencies are given in [0.1Hz] unit using 16-bit format. Voltage is a 8-bit value; 0 to 100% modulation index is described within the 0 to 255 range.
Duration	Values are measured with the CPU running at 25 MHz, without interrupt. ACM_GetCurrentStatorFreq: 100 μ s. ACM_GetCurrentVoltage: 1 μ s. ACM_GetCurrentSlip: 203 μ s.
Functions called	ACM_GetCurrentStatorFreq, ACM_GetCurrentVoltage, ACM_GetCurrentSlip, ACM_GetCurrentStatorFreq, IMC_GetCompare0, ACM_GetCurrentSlip, ACM_GetCurrentStatorFreq, IMC_GetRotorFreq.

ACM_InitSlipFreqReg

Synopsis	<pre>#include "acmotor.h" void ACM_InitSlipFreqReg(u8 OptimumSlip);</pre>
Description	<p>This function must be called before starting the regulation routine (typically after ACM_SoftStart).</p> <p>This function is used mainly to initialize the VoltageIntegralTerm routine, which is designed to smooth the transition from open loop operation (typically the end of start routine) to closed loop operations.</p> <p>It can also initialize the following routines (conditional compilation): SlipArray() and TrendArray().</p>
Inputs	OptimumSlip with [0.1Hz] unit.
Duration	Values measured with CPU running at 25 MHz, without interrupt. Maximum: 1.05 ms (USE_SMOOTHED_TREND switch off). Maximum: 1.12 ms (USE_SMOOTHED_TREND switch on).
Functions called	ACM_GetCurrentSlip, ACM_GetFuzzyError, ACM_GetCurrentStatorFreq, ACM_GetCurrentVoltage, slip_reg.
Caution	Since OptimumSlip has an u8 format, its value must be within the 0 to 25.5 Hz range.

ACM_SlipRegulation

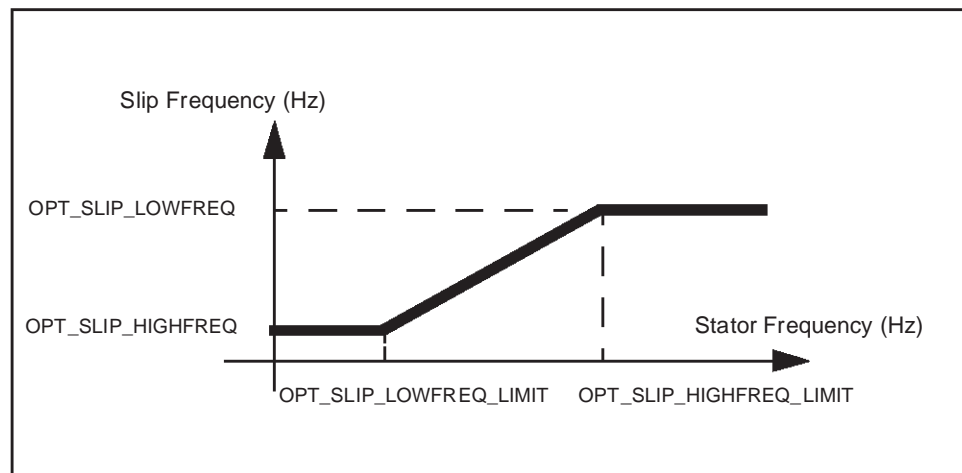
Synopsis	<pre>#include "acmotor.h" u8 ACM_SlipRegulation(u8 OptimumSlip);</pre>
Description	<p>This function performs a closed loop slip control to maintain the optimum voltage on stator windings.</p> <p>It can be shown that the overall AC motor drive efficiency is roughly a function of the slip frequency, having a maximum between zero and processed slip values.</p> <p>This function uses a fuzzy algorithm to compute the best voltage value using current and previous slip values (proportional and integral terms using standard naming conventions).</p>
Inputs	OptimumSlip with [0.1Hz] unit, in u8 format.
Returns	Modulation index voltage (u8 variable).
Duration	Values measured with CPU running at 25 MHz, without interrupt. Maximum: 1.4 ms (USE_SMOOTHED_TREND switch off). Maximum: 1.5 ms (USE_SMOOTHED_TREND switch on).
Functions called	ACM_GetCurrentSlip, ACM_GetFuzzyError, ACM_GetCurrentStatorFreq, slip_reg, ACM_VoltageMaxAllowed, plus ACM_SmoothSlip32 and ACM_SmoothTrend4 depending on conditional compilation (USE_SMOOTHED_TREND switch, see section 3.3.3.3 on page 58)
Caution	Since the OptimumSlip has an u8 format, its value must be within the 0 to 25.5 Hz range.
Code example	See "ACM_SlipRegulation" on page 58.
See also	See "Fuzzy Logic Regulation" on page 66. Flowchart (Figure 25. in Section 4.2.4).

ACM_GetOptimumSlip

Synopsis `#include "acmotor.h"`
 `u8 ACM_GetOptimumSlip(u16 StatorFrequency);`

Description The purpose of this function is to return the most appropriate slip frequency, based on the stator frequency, if this value changes within the motor operating range. Setpoints for this curve may be obtained either from the motor manufacturer or from empirical trials.

Figure 10. Optimum Slip Frequency



Inputs StatorFrequency with [0.1Hz] unit.

Returns OptimumSlip with [0.1Hz] unit, in u8 format.

Duration Maximum: 7.5 μ s (CPU running at 25 MHz, without interrupt).

Caution See ACM_VoltageMaxAllowed (same remarks on coefficients and data format).

ACM_GetFuzzyError

Synopsis	<pre>#include "acmotor.h" u8 ACM_GetFuzzyError (u16 SlipFreq, u8 OptimumSlip);</pre>
Description	<p>This function is used to first compute the difference between the current and the optimum (desired) slip frequencies and then to convert this signed result into Fuzzytech tool variable format, as follows:</p> <ul style="list-style-type: none">– Minimum value (negative) is 0,– Neutral value (no error) is 127,– Maximum value (positive) is 255. <p>This value is returned in the format to be used by the ACM_SlipRegulation function.</p>
Inputs	Current and optimum slip frequencies with [0.1Hz] format.
Returns	Unsigned variable which can be used by C fuzzy routines generated by the FuzzyTech tool.
Duration	Maximum: 2.8 μ s (CPU running at 25 MHz, without interrupt).

ACM_SmoothSlip32

Synopsis	<pre>#include "acmotor.h" u16 ACM_SmoothSlip32 (u16 SlipFrequency);</pre>
Description	<p>This routine calculates the rolling average of 32 consecutive values of the Slip frequency and can be used to cancel the high frequency component of the speed feedback (sensor dissymmetry, etc.).</p> <p>This will cutoff frequencies higher than $F_s/32$. F_s is the Sampling frequency (i.e. the number of calls to this function per second) and must be stable.</p>
Inputs	SlipFrequency with [0.1Hz] unit.
Returns	Rolling average is returned with [0.1/32] Hz (i.e. raw sum of the 32 previous inputs).
Duration	Maximum: 68 μ s (CPU running at 25 MHz, without interrupt).
Note	<p>Using [0.1/32] Hz has the advantage of preventing definition losses.</p> <p>The returned value can either be used as an input for the ACM_SmoothTrend4 function or divided by 32 to obtain a smoothed slip frequency with [0.1Hz] unit.</p> <p>This function will only be available if the USE_SMOOTHED_TREND switch is ON (conditional compilation) see Section 3.3.3.3.</p>
Code example	ACM_InitSlipFreqReg and ACM_SlipRegulation function can be consulted for the practical implementation of the ACM_SmoothSlip32 function.

ACM_SmoothTrend4

Synopsis	<pre>#include "acmotor.h" s8 ACM_SmoothTrend4 (u16 PreviousSmoothedSlip, u16 SmoothedSlip);</pre>
Description	<p>This routine calculates the rolling average of 4 consecutive trend values (difference between two consecutive Slip values) and returns a smoothed trend value.</p> <p>Using previously filtered Slip values and performing an additional average is required to extract the real speed trend and to minimize the high pass effect amplifying parasitic speed feedback noise.</p>
Inputs	Both PreviousSmoothedSlip and SmoothedSlip are using the output format of the ACM_SmoothSlip32 function (i.e. $[0.1/32]$ Hz).
Returns	Signed value with $[(0.1 / 32) / (4 * x)]$ Hz/ms] (x being the time between two calls to ACM_SmoothTrend4). The returned value can be used as it is, or as an input to any regulation algorithm (PI, PID, Fuzzy, etc.) which must take into account the particular format.
Duration	Maximum: 75 μ s (CPU running at 25 MHz, without interrupt).
Note	<p>The user must keep in mind that returned value units (and therefore reliability) depend on the timebase used to call this function.</p> <p>This must especially be taken into account for the first call to this function (required latency before any use of these values).</p> <p>This function will only be available if the USE_SMOOTHED_TREND switch is ON (conditional compilation). See Section 3.3.3.3.</p>
Code example	The ACM_InitSlipFreqReg and ACM_SlipRegulation functions can be consulted for the practical implementation of the ACM_SmoothTrend4 function.

3.3.3 Detailed Explanations and Customization

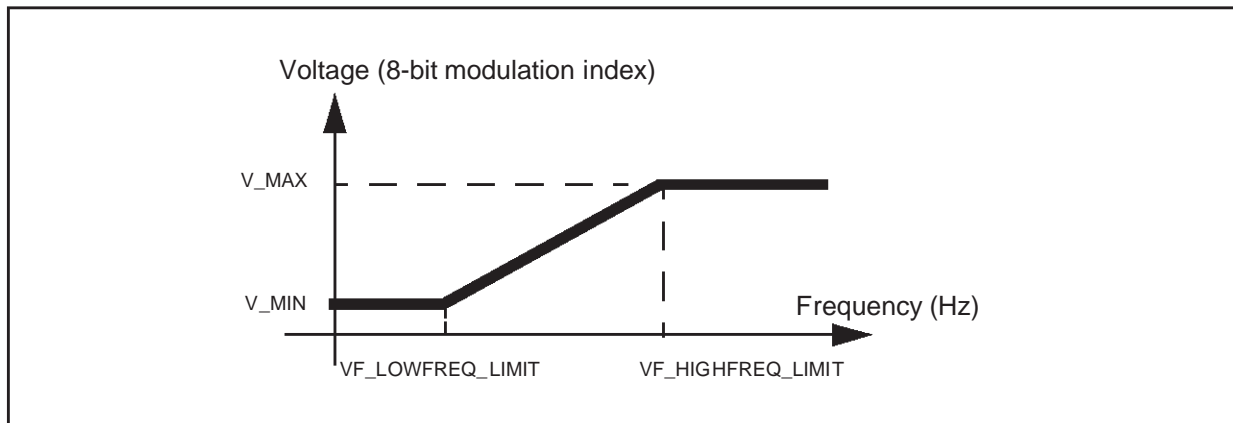
3.3.3.1 ACM_VoltageMaxAllowed

The stator windings of an AC motor can be approximately first represented as an inductance, for which the impedance will therefore increase as the stator frequency becomes higher.

The scalar control method is commonly used with AC motor drives to assume that torque will be maintained by keeping a constant voltage versus frequency ratio.

Above a certain frequency limit, this ratio will decrease as the voltage is limited by the inverter topology, that is practically reached with 100% modulation (i.e. voltage = 255). A minimum voltage must also be maintained at low speed in order to energize the stator windings. (See Figure 11.).

Figure 11. Practical Implementation of Standard V/f Characteristics



Voltage is calculated by the following equations:

$$\text{Voltage} = \frac{(\text{StatorFrequency} \times \text{VF_COEFF}) - \text{VF_OFFSET}}{256} + \text{V_MIN}$$

$$\text{VF_COEFF} = \frac{(256 \times (\text{V_MAX} - \text{V_MIN}))}{(\text{VF_HIGHFREQ_LIMIT} - \text{VF_LOWFREQ_LIMIT})}$$

$$\text{VF_OFFSET} = \text{VF_COEFF} \times \text{VF_LOWFREQ_LIMIT}$$

Setpoints can be entered in the acmparam.h header file. Both VF_COEFF and VF_OFFSET will be re-computed at compile time (multiplication by 256 is used here to maintain a sufficient accuracy and to decrease quantification effects).

Important: All variables used in this function are 16-bit. When modifying setpoints, the user must verify the following:

$$\text{VF_OFFSET} \leq 0xFFFF$$

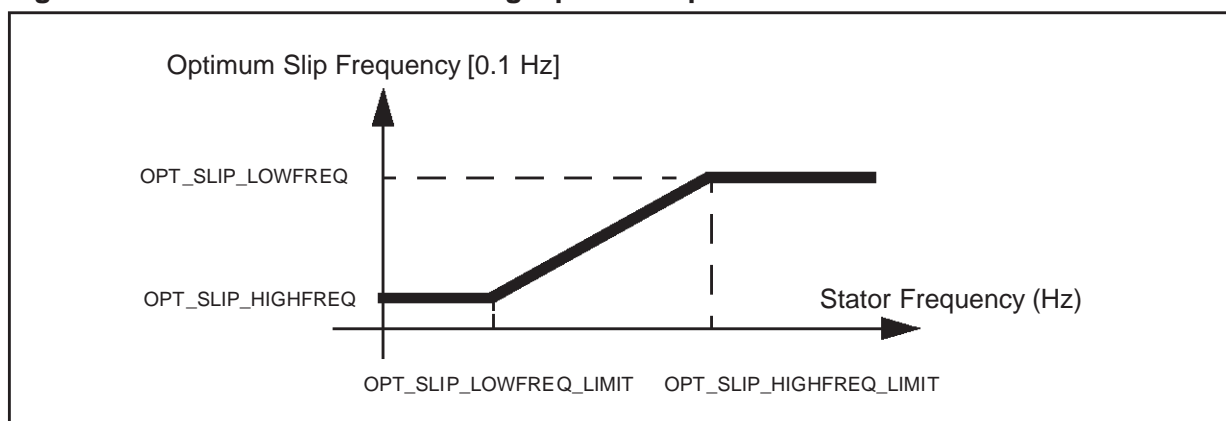
$$((\text{VF_HIGHFREQ_LIMIT} \times \text{VF_COEFF}) - \text{VF_OFFSET}) \leq 0xFFFF$$

Otherwise, **certain variables and constants may have to be declared as u32 instead of u16.** (Buffer declared inside ACM_VoltageMaxAllowed, VF_OFFSET, etc.).

3.3.3.2 ACM_GetOptimumSlip

The purpose of this function is to return the most appropriate slip frequency, based on stator frequency, if this value changes inside the motor operating range. Setpoints for this curve may be obtained either from the motor manufacturer or from empirical trials (see Figure 12.).

Figure 12. Linear Function returning Optimal Slip Value



The equations used to obtain the optimum slip value are comparable to the value described for the ACM_VoltageMaxAllowed function (see Section 3.3.3.1).

Setpoints can be entered in the acmparam.h header file. Both SLIP_COEFF and SLIP_OFFSET will be re-computed at compile time (multiplication by 256 is used here to maintain a sufficient accuracy and to decrease quantification effects).

Important: All variables used in this function are 16-bit. When modifying setpoints, the user must verify the following:

$$\text{SLIP_OFFSET} \leq 0\text{xFFFF}$$

$$((\text{OPT_SLIP_HIGHFREQ_LIMIT} \times \text{SLIP_COEFF}) - \text{SLIP_OFFSET}) \leq 0\text{xFFFF}$$

Otherwise, **certain variables and constants may have to be declared as u32 instead of u16.** (Buffer declared inside ACM_GetOptimumSlip, SLIP_OFFSET, etc.).

3.3.3.3 ACM_SlipRegulation

Since ACM_SlipRegulation uses a fuzzy engine to compute the most appropriate voltage from the current slip value (see Section 3.5), few functions can be customized:

- The regulation loop time may be modified to adjust the bandwidth. Nevertheless, since an accumulative term “VoltageIntegralTerm” is used in this routine (the PI integral like term), increasing loop time will decrease its effects (accumulation will be slower and integral action on the output will be delayed). Inversely, decreasing loop time will increase its effects (accumulation will be faster and integral action on the output will be increased).
- A USE_SMOOTHED_TREND commutator may be switched ON to filter one of the slip_reg “Error_slip_reg” input variables. Refer to the description of the ACM_SmoothSlip32 and ACM_SmoothTrend4 functions.

Below is an example of the use of this regulation process.

```
ACM_SoftStart(StartFrequency, MinTachoFrequency); /* Open loop*/

OptimumSlip = ACM_GetOptimumSlip(ACM_GetCurrentStatorFreq());
ACM_InitSlipFreqReg(OptimumSlip); /* Mandatory regulation initialization */

IMC_Load_Reg_LoopTime(10); /* Regulation time base set-up */

/* From here regulation can be called periodically */

...
...

while (NoStopOrderReceived()) /* Example of regulation stop event */
{
    if (IMC_Reg_LoopTime_Elapsed()) /* Periodic re-start */
    {
        OptimumSlip = ACM_GetOptimumSlip(NewFrequency);
        NewVoltage = ACM_SlipRegulation(OptimumSlip);
    }
}
```

Note: ACM_SoftStart is considered as an open loop because the tacho feedback is just used at that specific time to limit the voltage when the rotor shaft turns at a fast enough speed.

It is recommended to exit this routine with a slip that is close to or slightly lower than the slip to be regulated for smoothing the transition to closed loop operations (to be determined empirically).

Two functions must be called before starting the regulation process:

- ACM_InitSlipFreqReg to initialize the Integral like term.
- IMC_Load_Reg_LoopTime to set-up the regulation time base.

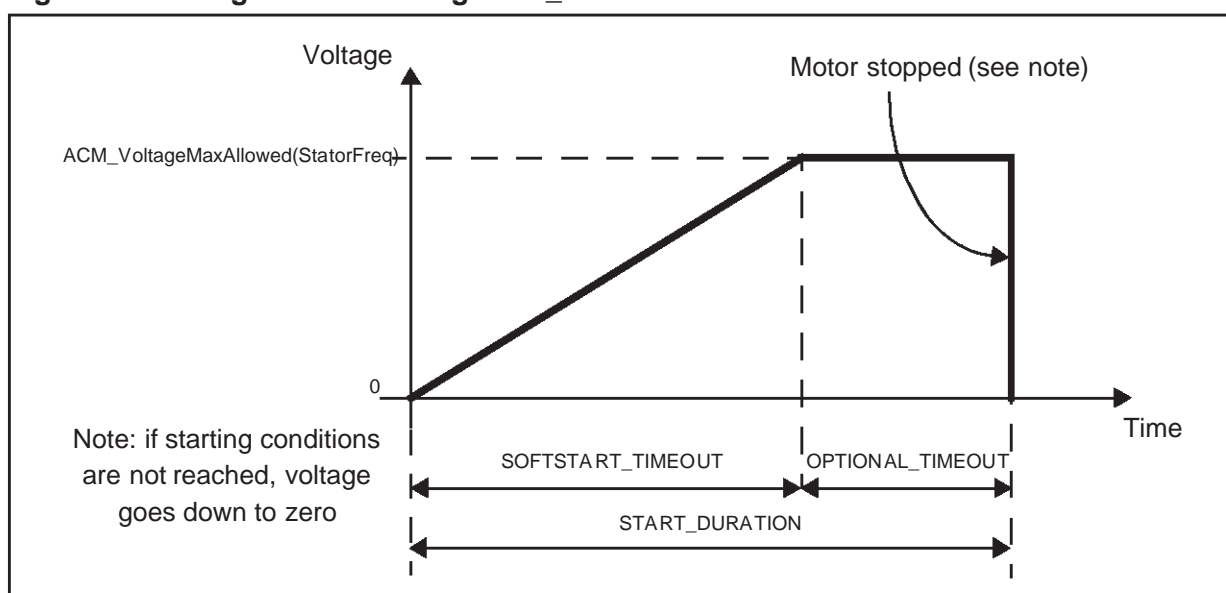
3.3.3.4 ACM_SoftStart

This Soft Start function basically imposes a voltage profile to start the motor without an excessively heavy inrush current. This profile, which can be parameterized depending on the application, is described in Figure 13..

Two parameters can be set (SOFTSTART_TIMEOUT and START_DURATION) while the third one (OPTIONAL_TIMEOUT) is computed at compile time. These constants are located in the acmparam.h header file.

OPTIONAL_TIMEOUT, which is sometimes mandatory for high inertia loads, may be cancelled by setting SOFTSTART_TIMEOUT equal to START_DURATION.

Figure 13. Voltage Profile during ACM_SoftStart Routine



A flowchart is also given in Figure 22. in Section 4.2.1.

3.3.3.5 Using P Pole Motors

The current library has been setup for an AC motor with a single pair of poles (unipolar), where the stator and rotor frequencies are directly comparable and the slip can be calculated in a simple way (subtraction).

Generally (motors with p pair of poles) the following relations apply:

$$F_{\text{rotor}} = \frac{F_{\text{stator}}}{p}$$

$$\text{Electrical slip: } F_{\text{slip}} = \frac{F_{\text{stator}}}{p} - F_{\text{rotor}}$$

$$\text{Pseudo slip: } F_{\text{slip}}' = F_{\text{stator}} - p \times F_{\text{rotor}}$$

In order to make setting parameters easier, only the pseudo slip will be taken into consideration. This means that value returned by `ACM_GetCurrentSlip` will be p times the real electrical slip frequency.

The user will therefore have to keep in mind this definition when:

- measuring the rotor speed via the `IMC_GetRotorFreq` function, whose return is also p times the real rotor speed,
- characterizing its motor (efficiency vs slip characteristics),
- setting-up `ACM_GetOptimumSlip` parameters.

Parameters are customized in the `IMCParam.h` file, by modifying the `MOTOR_POLES_PAIR` constant used in the `IMC_GetRotorFreq` function to calculate the frequency.

3.3.3.6 Stator Frequencies above 340 Hz

Note: This section is obsolete when using Version 2.0 where stator frequencies can go up to 680 Hz without any customizing or voltage loss. Please refer to application note AN1277 for more details.

The ACMotor module is parameterized to limit the stator frequency to 339.4 Hz, which correspond to 20,000 RPM for a unipolar motor.

A higher stator frequency may be required when using very high speed or multi-polar motors. Therefore, the acmparam.h module can be customized in order to be able to work with frequencies up to 680 Hz (approximately 40,000 RPM for a 4-pole motor).

This is achieved by re-defining the following constants and tables:

- HIGHEST_FREQUENCY (in acmparam.h),
- LOWEST_PERIOD (in acmparam.h),
- Sin_ref_tab_12[3] (in acmparam.h),
- Sin_ref_tab_48[12] (in acmparam.h),
- Mean_sine (in sine.asm)
- Max_sine (in sine.asm)

Default values are set for a maximum operating frequency of 339 Hz. The set of values for 680 Hz are commented (in both acmparam.h and sine.asm).

It is recommended that the 680 Hz operating frequency be used only if necessary because this will result in a 5% loss of motor voltage due to variable PWM frequency operation. This drawback may be eliminated by modifying the reference sine wave quarter tables (Sin_ref_tab_12 and Sin_ref_tab_48) and adding a third harmonic to the fundamental. In this case, the sine coefficient MUST NOT exceed the maximum value in the reference tables (See Table 3).

Table 3. Reference Sine Wave Quarters and Absolute Max. Values for Customization

Max. Frequency	Table Name	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10	Step 11	Step 12	Max
339.4 Hz	Sin_ref_tab_48	16	48	78	108	136	161	183	203	219	231	239	243	243
339.4 Hz	Sin_ref_tab_12	65	178	243										243
680 Hz	Sin_ref_tab_48	15	45	74	102	128	152	173	192	207	218	226	230	230
680 Hz	Sin_ref_tab_12	102	192	230										230

WARNING : Never run a motor above its rated stator frequency. This may cause the rotor or ball bearings to explode if the speed is higher than the absolute maximum rating.

3.4 ADC MODULE

3.4.1 Purpose

This module (ADCModul.c) is used for the following functions:

- initialize and start the A/D converter,
- obtain the ready-to-use values from the converter,
- monitor the performance of the ADC peripheral in the main routine.

It was basically written to monitor signals that vary slowly, such as trimmers, since the sampling rate is set to 1 acquisition per 530 μ s.

3.4.2 Description

The ADC is initialized in Continuous mode: When all channel conversions are completed (from Ch2 to Ch7), an End Of Conversion Interrupt is issued for processing this data and channel scanning is automatically restarted.

The Filtering method uses a classical average: Once a predefined number of acquisitions is completed, their result is stored in a buffer and their average value is calculated by division.

3.4.3 Synopsis

```
#include <ADCModul.h>
...
ADC_Init();
...
CORE_Enable_Interrupts();
...
ADC_Start();
...
    /* From this point, you can call any time functions
    ADC_Get_Chx(); x being the number of the channel to be read */
...
{
    u8 SpeedTrimmer, VoltageTrimmer;

    SpeedTrimmer = ADC_Get_Ch2();
    VoltageTrimmer = ADC_Get_Ch3();
}
```

3.4.4 Memory Use

All variables are stored in RAM for easier portability.

Each channel requires 3 bytes (2 for buffers and 1 for results).

Number of averages counter: 1 byte.

Watchdog counter: 1 byte.

Current version: *20 bytes*

3.4.5 Timings

The ADC input clock is prescaled here so that conversions are slowed down (to lower the CPU load).

Each conversion is achieved in 138 clock cycles. This means that the following is required in order to complete 6 conversions with an input clock prescaler set to 16:

$T = 138 \times \text{Number of channels} \times \text{Prescaler factor} \times (1 / \text{INTCLOCK}).$

$T = 138 \times 6 \times 16 \times 40 \text{ ns} = 530 \text{ }\mu\text{s}.$

T is the average duration between End Of Conversion interrupts.

The interrupt duration is approximately 14 μs (average), the maximum time being 43 μs (outside nested interrupts) with the CPU running at 25 MHz.

3.4.6 Software Watchdog

The ADC_WDT_Counter variable is increased (up to 255, without roll-over) every time an End Of Conversion interrupt (EOC_IT) is issued.

The purpose is to monitor the number of interrupts that have been issued between 2 main loop scanning periods (assuming that this period is roughly constant and known) in the main program.

If this number exceeds a predefined range, an error has occurred (the ADC has been stopped, an external interrupt is overloading the CPU, etc.). The user is then free to either correct the error or to let the hardware watchdog reset the MCU.

This ADC_WDT_Counter variable can only be accessed through function calls so as to have a Read-only like variable which cannot be modified erroneously by other modules.

3.4.7 Caution

The following important points must be taken into consideration.

3.4.7.1 ADC Register Declaration File Version

The header file containing the ADC register and bit declarations used by the C compiler must be release 5.0 of *ad_c.h* and must be located in the include.st9 directory of the toolchain (V4.3 release).

This include file contains the declaration for the new registers needed for this type of ST9+ 8-bit ADC with input clock prescaler.

3.4.7.2 Sampling Issues

The maximum frequency for input signals can be calculated as a function of the conversion speed and number of acquisitions needed before updating the averaged value.

Example: The time required to scan the channels is 530 μ s and since 4 values are needed for averaging, the total time between data updates is $530 \mu\text{s} \times 4 = 2.12 \text{ ms}$.

Shannon's theorem will enable us to monitor signals of up to approximately 230 Hz (i.e. $1/(0.00212 \times 2)$).

3.4.7.3 Value Availability Time

The user must reserve a certain amount of time until the first averaged values become available (2.12 ms after *ADC_Start()*; in the previous example).

3.4.7.4 Interrupts

Interrupts are used in Nested mode. The priority level for ADC interrupts is set to 6.

3.4.7.5 Port Initialization

The I/O ports corresponding to the ADC channels must be previously initialized as Analog Inputs in another module (for example, in an *IO_AnalogPorts_Init* function).

3.4.8 Customizing the ADC Module

This module is just an example of ADC use. The user can perform the following actions depending on the application:

- Decrease memory needs by not scanning all 7 channels (keep in mind that Auto Scan mode requires that the scanning procedure ends with channel 7). Function prototypes, variable declarations and interrupt service routines must be changed this way (i.e. basically by deleting unused code).
- Increase the speed of channel scanning by decreasing the ADC Clock prescaler.
- Increase the speed of the interrupt service routine by using registers from the register file rather than from RAM locations.

- Set the ADC to Single Shot mode and have it start at special events (for example: IMC_User_1ms_Routine, see ZPC_IT chapter of IMC module.
- Use the Analog watchdog functions (available only on channels 6 and 7) to decrease the CPU load.

Ex: Once a trimmer has been read, the Analog watchdog thresholds can be set close to the measured value and the End Of Conversion interrupts can be disabled. Once the trimmer value has changed and has it has reached the predefined thresholds, Analog Watchdog interrupts will be issued.

For further explanations and details, see also:

- ADCModul.c and ADCModul.h files in-line comments,
- ST92141 Datasheet.
- ST9+ User Guide

3.5 FUZZY LOGIC REGULATION

3.5.1 Description

The benefits of using fuzzy logic when designing embedded control systems have already been proven in a wide variety of applications. The ease and speed of their implementation as well as their higher flexibility in regards to a classical PID algorithm (with equivalent development time) were the key selection criteria for this project.

The usual drawbacks, memory use and execution time, are not critical for the ST92141. For reference, the fuzzy engine only required about 1 Kbyte of ROM (including the fuzzy kernel), 45 bytes of RAM and is completed in less than 0.9 ms.

The fuzzy logic regulation was designed using the Inform Software Corporation *fuzzyTECH* tool, a convenient way to obtain a ready-to-compile file from graphical settings. This C-precompiler generates ANSI C code traducing project data: fuzzy membership functions set-points and rules. Even if C code can be compiled on any host platform, the *fuzzyTECH* 5.2 MCU-C Precompiler Edition is recommended as it is especially suited for 8/16-bit MCUs due to its efficient data structures and MCU-optimized programming style.

Two files, `slip_reg.c` and `slip_reg.h`, were pre-compiled from the `slip_reg.ftl` project (Fuzzy Tech Language) and contain 2 functions that are accessible from the user modules:

- `initslip_reg`: Initializes the fuzzy engine and must be called only once before closed-loop operations,
- `slip_reg`: Contains the fuzzy engine itself. Input and Output variables are global (`Error_slip_reg` and `Frequency_slip_reg` are inputs, `Voltage_slip_reg` and `DeltaVoltage_slip_reg` are outputs).

The following two generic files (application independent) must also be included in the working directory in order to use the fuzzy logic regulation:

- `ftc8.l`: Contains the fuzzy kernel (i.e. a set of generic fuzzy functions used for fuzzification, rule execution and defuzzification). This file, located in the `/fuzzylib` folder of the working directory contains the object files archived with the AR9 utility (see Section 3.5.3).
- `ftclib.h`: Contains some typedef and fuzzy kernel function prototypes.

3.5.2 Fuzzy Engine Technical Characteristics

The fuzzy control algorithm uses 2 inputs and 2 outputs and is characterized by a set of 21 rules. The centre of maximum method is used to defuzzify the fuzzy inference. As mentioned above, it is executed in less than 0.9 ms.

Fuzzy variables are:

- current stator frequency: Frequency_slip_reg input, described with 3 membership functions (mbf),
- slip frequency error: Error_slip_reg input, described with 7 mbfs,
- voltage: Voltage_slip_reg output, described with 5 mbfs,
- voltage variation: DeltaVoltage_slip_reg output, described with 6 mbfs,

This last term is accumulated to take into account the “history” of the slip frequency error. It balances positive and negative portions to eliminate steady state errors in the same way that a PID algorithm does. It is then added with the voltage term to give the final control output.

3.5.3 Customization

3.5.3.1 Updating the ftc8.l Fuzzy Kernel Library

The fuzzy kernel is delivered with the *fuzzyTECH* pack as several C source files. They need to be compiled in a object library with the same options as the C sources used in the project. A batch file included in the utility folder (stmake8.bat) is available for re-compiling the ftc8.l file from these C sources. Further details on customization may be found in the appendix, Section 4.4.

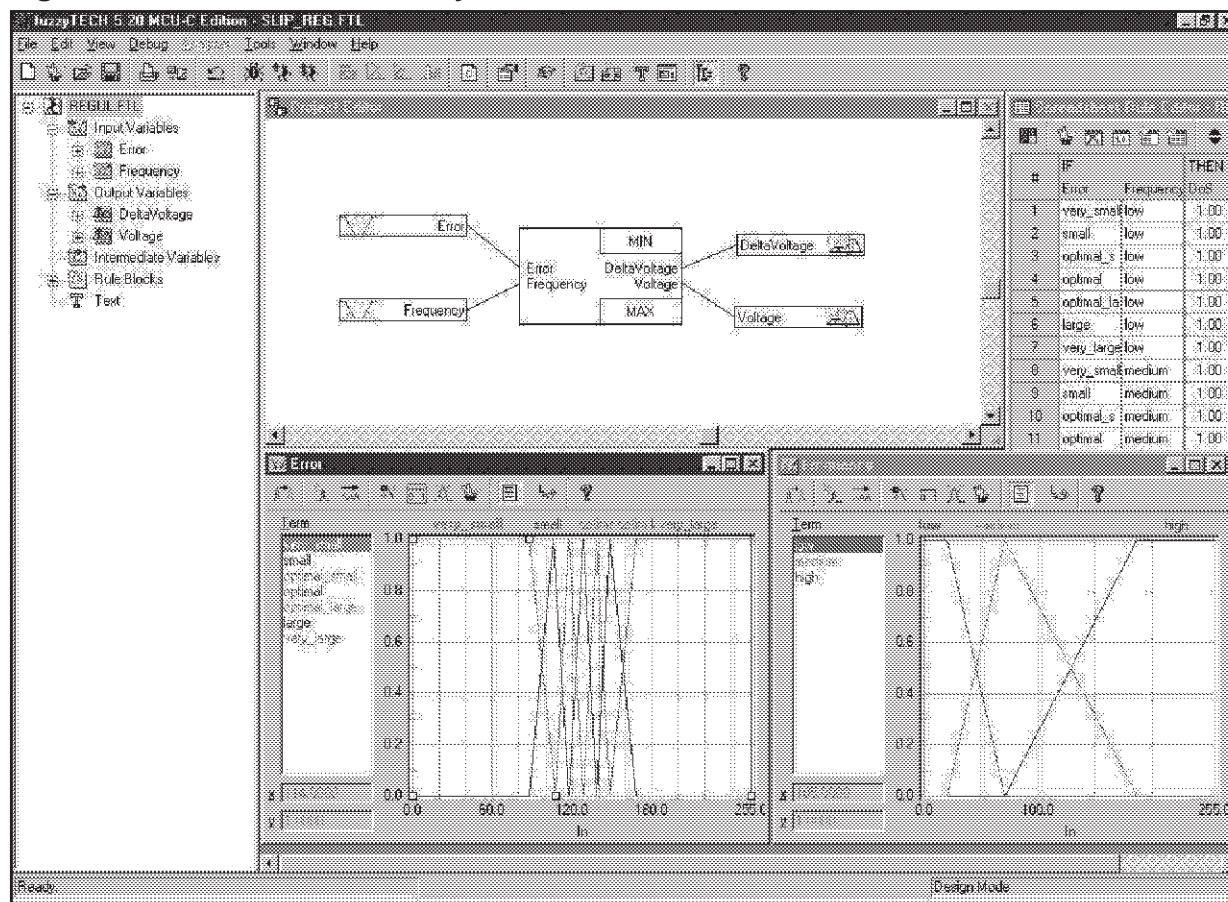
3.5.3.2 Modifying the Fuzzy Engine

The *fuzzyTECH* software must be purchased in order to be able to modify the fuzzy engine. All information can be found on the Internet at <http://www.fuzzytech.com>.

The project was pre-compiled using the *fuzzyTECH* 5.2 MCU-C Precompiler Edition (see Figure 14.). The current release is “*fuzzyTECH* 5.31d MCU-C Edition”

The project source file “slip_reg.ftl” is available on request in order to allow users to customize and improve the delivered algorithm.

Figure 14. Inform Software fuzzyTECH Tool



3.6 I/O MODULE

3.6.1 Description

The purpose of this module is to keep all information concerning the ports within the same file. This will make sharing I/Os between the peripherals and/or the interfacing functions more clear.

I/Os are initialized at the beginning of the main program, just after the PLL start.

A single function (IO_Init_Ports) is externally accessible. Peripheral or module I/Os (SPI, Analog, Push-buttons, LED, etc.) must be initialized separately in static functions (see module in-line comments).

3.6.2 I/O Access

Once the ports are initialized, the P3DR and P5DR data registers can be directly accessed anywhere in the program without using a paging mechanism, since they are located in group E below R240 (respectively R227 and R229), using the DPRREM bit set to 1 in the EMR1 register of the MMU.

For further explanations, see also ST92141 Datasheet, I/O ports chapter.

3.7 RCCU MODULE

3.7.1 Description

This module is used for the Reset and Clock Control Unit. It is currently used only for the PLL initialisation, using the following settings:

- 5 MHz crystal, divided by 2 before the PLL to obtain an accurate 50% duty cycle.
- Multiplication by 10 to obtain a 25 MHz CPU clock.

3.7.2 Customization

Outside functions provided with ST9+ standard library, this module should contain all functions related to:

- Slow mode (SLOW1, SLOW2),
- Wait for Interrupt (WFI),
- Low Power WFI.

3.8 CORE MODULE

3.8.1 Description

The purpose of this module is to group together functions related to the ST9+ Core.

It mainly deals with interrupts, since most of the system registers are already initialized in the crt9.asm start-up file.

3.8.2 Available Functions

■ CORE_Init_External_Interrupt

This function is called during MCU initialisation. The external interrupt vector base address must correspond to the address used in the crt9.asm file. Interruption mode is Nested and the external interrupt priority is not set (reset value is 7, i.e. they will not be acknowledged in Nested mode).

■ CORE_Enable_Interrupts / CORE_Disable_Interrupts.

Both functions apply to interrupts received from either external or peripheral sources.

3.8.3 Customization

3.8.3.1 DIV_Zero_Trap

This trap routine must be filled in by the user to determine the behaviour of the program when there is a divide-by-zero attempt. This may occur during an assembly routine (calc_rpt_tab uses div mnemonic) or during division operations from the set of mathematical routines (stdlib.h, math.h, etc.) provided with the C compiler (see relevant literature).

3.8.3.2 External Interrupts

The user can include external interrupt subroutines in this module under the following conditions:

- the interrupts are enabled with the expected edge sensitivity,
- the interrupts have a priority strictly superior to 7 (due to Nested mode),
- the corresponding port has been properly set-up.

3.8.3.3 NMI Interrupt

The NMI interrupt routine is declared inside this Module. The user must enter the executable code.

Due to the particular behaviour of the NMI pin on the ST92141 (sets all PWM outputs to high impedance if properly set-up), all data referring to its use must be carefully read in the datasheet, especially in the following sections:

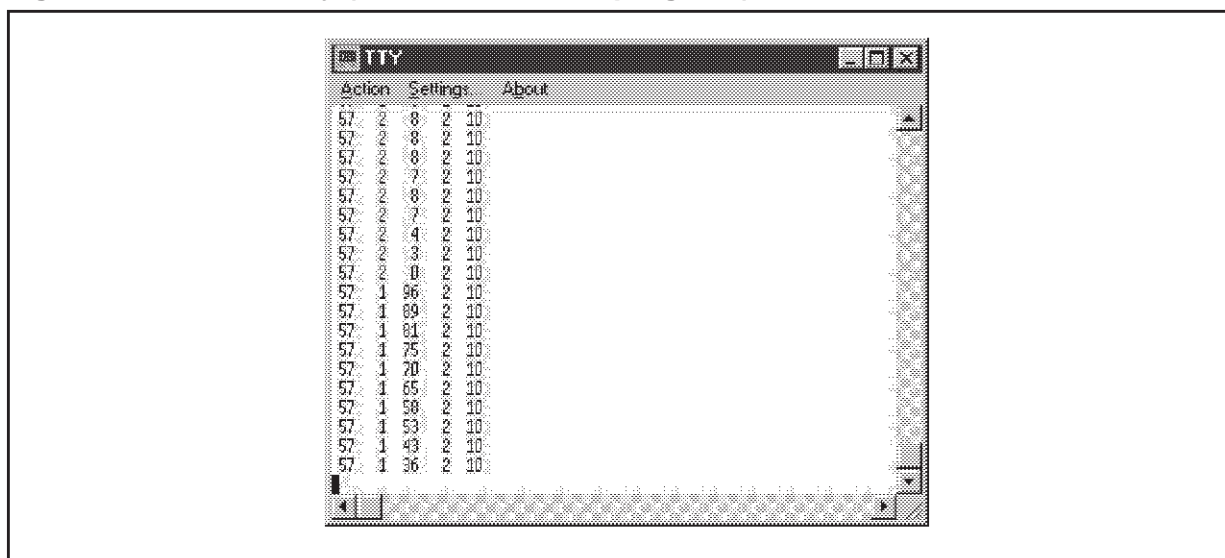
- INTERRUPT section, Top level interrupt chapter
- INTERRUPT section, NMI / WKPO line management chapter
- IMC section, NMI management chapter.

3.8.4 UART Module

3.8.5 Description

This module contains a software-emulated UART that is used to upload data on a PC. When used with the appropriate software terminal (tty_5 utility delivered with the library) it is a very convenient tool that can monitor, trace and store MCU internal data in the application without using the debugger. The hardware resource for this UART is the ST9+ standard timer. Bytes are output using interrupts to reduce the CPU load and the priority is set to 5 (i.e. lower than the motor control peripheral, but higher than the ADC).

Figure 15. TTY_5 Utility (Data download in progress)



3.8.6 ST92141 Characteristics

Data to be sent must be stored in the Transmit_data[0...4] array. Once the SendDataToPC function has been called, the content of this array will be output on the P3.3 port, with the following characteristics:

- 9600 bauds
- 8-bit data
- 1 stop bit
- No parity
- hexadecimal format (ASCII terminal cannot be used with the current UART version)
- No communication protocol

Note: Any attempt to restart a transmission while the previous one is not completed will be cancelled.

This UART works with standard timer interrupts and the priority for this peripheral is set to 2.

3.8.7 PC Characteristics

Once the `tty_5` executable has been launched (all default settings are correct, only the port has to be checked), data will be displayed in real time at a rate of 5 bytes every 10 ms and simultaneously written in a `data.txt` log file located in the `tty_5` working directory.

Since the ST9 is Master during the uploading sequence, the transmission timebase is reliable and data files can be imported on any spreadsheet editor, e.g. for closed loop performance evaluation.

3.8.8 Customization

The communication baud rate can be adjusted from 1200 up to 38400 bauds, keeping in mind that the interrupt processing load will increase accordingly (ten interrupts being issued for every byte transmitted). The rest of the protocol remains the same (1 stop bit, no parity, etc.).

3.8.9 Important Notice for Hardware Implementation

There must be a galvanic isolation between the ST92141 and the PC serial port if there is none available between the ST92141 and the power inverter. This will lower the risk of user injury and computer destruction.

This is achieved by adding opto-isolators between the MCU and the level converter (RS 232 transceiver) directly connected to the serial bus.

Two-lines are needed for asynchronous transmission; the pin number corresponds to a standard DB9 female connector:

- Tx (pin 2 for straight cable, pin 3 for null modem cable).
- Ground (pin 5 of DB9 female connector).

3.9 CODE EXAMPLE

Two demo programs are included in the main.c routine. Two define statements are used to choose either Open Loop (OPEN_LOOP) or Closed Loop (CLOSED_LOOP) operations.

The following procedure may be used for starting the practical implementation:

- Test the open loop operation without using the power inverter to verify the PWM signals (sine waves can be displayed using a 10K / 22nF RC low-pass filter) and to validate any UART links.
- Test the open loop operation with the power inverter by slowly increasing the DC bus voltage to verify the power stage and to define the best "Vmax versus frequency" curve for the motor. This stage can also be used to validate the correct tacho frequency measurements via the UART.
- Finally, test the closed loop operations and system stability.

Both modes are described below.

3.9.1 Open Loop: Voltage and Frequency are individually adjustable

This program converts ADC readings (channels 2 and 3 are scanned every 100 ms) into voltage (0 to 255 modulation index) and stator frequency settings (0.9 to 255 Hz).

Since no tests are performed on the V/f ratio, trimmers (multi-turn -type trimmers are recommended) must be handled carefully, starting from low voltage/ low frequency. In any case, if the start-up voltage is greater than 10, it is slowly increased until it reaches its nominal value to prevent an excessive inrush current.

Note: Both NewVoltage and NewStatorFreq variables are declared in the register file for demonstration purposes only. In Version 2.0, both variables are declared in the main.c file.

3.9.2 Closed Loop

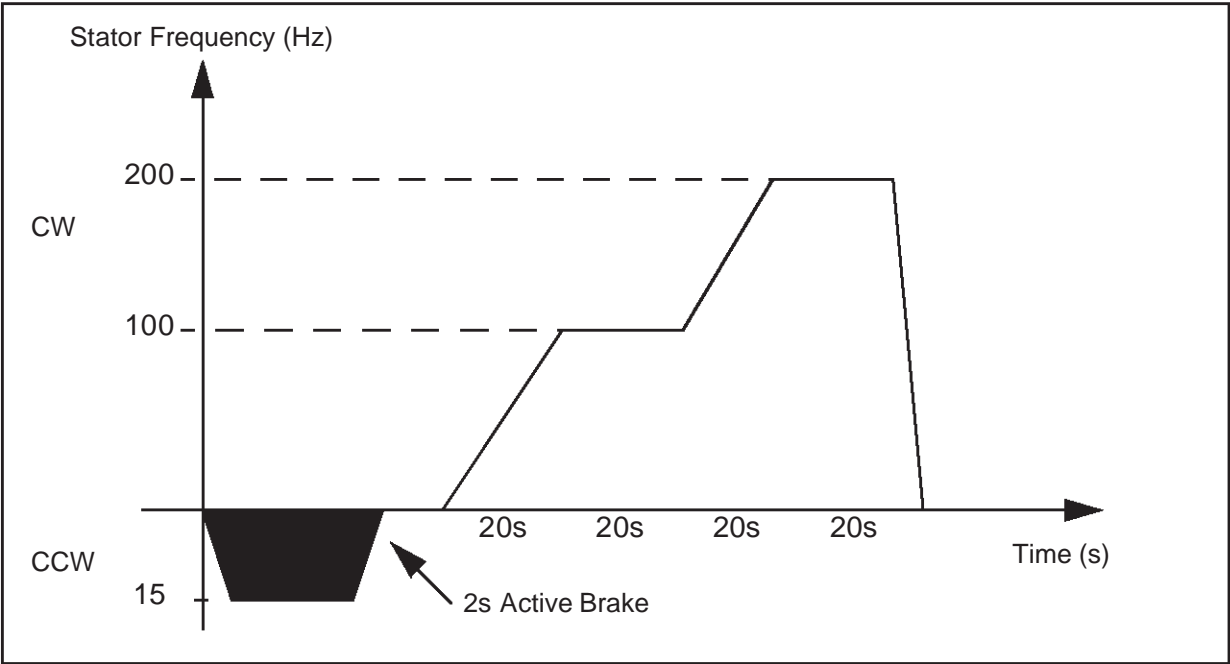
Tacho parameters have been set for 8 pairs of poles for the tacho and 1 pair of poles for the motor.

The implemented speed profile is shown in Figure 16. below. The target speed is calculated by subtracting the regulated slip from the given stator frequency (set in the profile).

Example: At 100Hz, given an optimum slip frequency of 2 Hz, the rotor speed will be 5,880 RPM (98 Hz).

This profile may be easily modified to characterize both regulation (accuracy, speed response) and motor (torque and acceleration capability) characteristics.

Figure 16. Closed Loop Speed Profile



3.10 DESIGNING WITH 92141LIB: LIBRARY INTEGRATION

Once the tools and demo programs have been successfully run, the user will have to integrate this library in his design. Some modules will be used as they are, others will have to be filled in and some will have to be created.

Before starting this process, the following two important points must be highlighted:

- upgrades (various improvements, new modules or modulation methods, etc.)
- support for tools and/or application software (even though source codes and flowcharts are provided to make customization and in depth library understanding as easy and clear as possible).

Note: Upgrading between Version 1.0 and Version 2.0 is described in detail in application note AN1277.

Users should follow the recommendations listed below in order to very easily benefit from the latest library upgrades (by simply replacing existing modules with the new ones without any source level modifications):

- Do not modify either of the two IMCModul.c and ACmotor.c modules. These modules can be customized in the IMCParam.h and ACMPParam.h header files (see Table 4 for other modules).
- Carefully trace all modifications to generic modules in order to make support easier.
- Regularly monitor the stack during development (at least when adding routines which increase nesting depth). The same should be done for both ROM/RAM use (data available as a percentage in file acmotor.map, generated at compile time).
- Maintain existing function interfaces.
- A modular approach should be used as often as possible.
- Use functions such as ACM_SoftStart, ACM_SoftStart_OpenLoop, ACM_RampUp or ACM_SustainSpeed, for example, to implement modified versions outside source modules if they do not fit targeted applications.

Note: ACM_RampUp and AMC_SustainSpeed functionsq have been slightly modified in Version 2.0. Please refer to application note AN1277 for more details.

The following matrix (Table 4) describes the module classification for customization:

- Class A: The code inside the module can be partially removed,
- Class B: The code can be added to a module. Existing functions, define statements and constants should nevertheless be maintained for upward compatibility,
- Class C: Modules must not be modified (start-up files, SW library core modules, automatically generated source code).

Table 4. Module Classification (cf. above text for explanations on A, B, C classes).

			Module Name
A			acmotor.c & .h
B	x		acmparam.h
C	x		adcmodul.c & .h
	x		core.c & .h
x			crt9.asm
		x	debug.h
	x		define.h
x			ftclib.h
x			IMCModul.c & .h
	x		IMCparam.h
	x		IOModule.c & .h
		x	main.c
	x		makefile
	x		RCCU_Mod.c & .h
	x		Reg_file.h
	x		sine.asm
x			slip_reg.c & .h
	x		UART.c & .h

4 APPENDIX

4.1 IMC MODULE FLOWCHARTS

4.1.1 Automatic Data Transfer Interrupt (ADT)

Note: ADT_IT routines are processed differently in Version 2.0. Please refer to application note AN1277 for more details.

Figure 17. Main Task (Pointer Update and Compare Register Refresh)

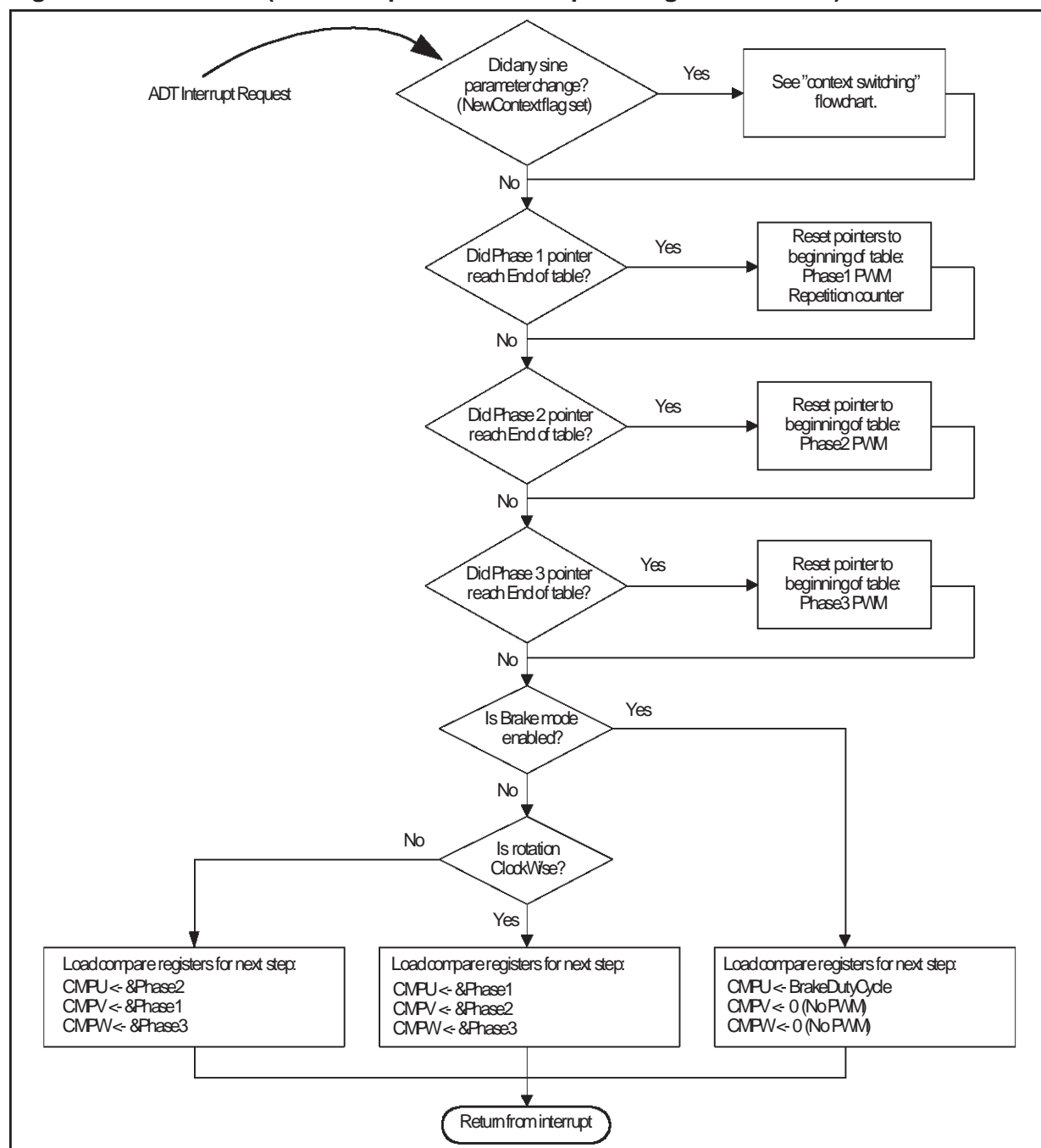
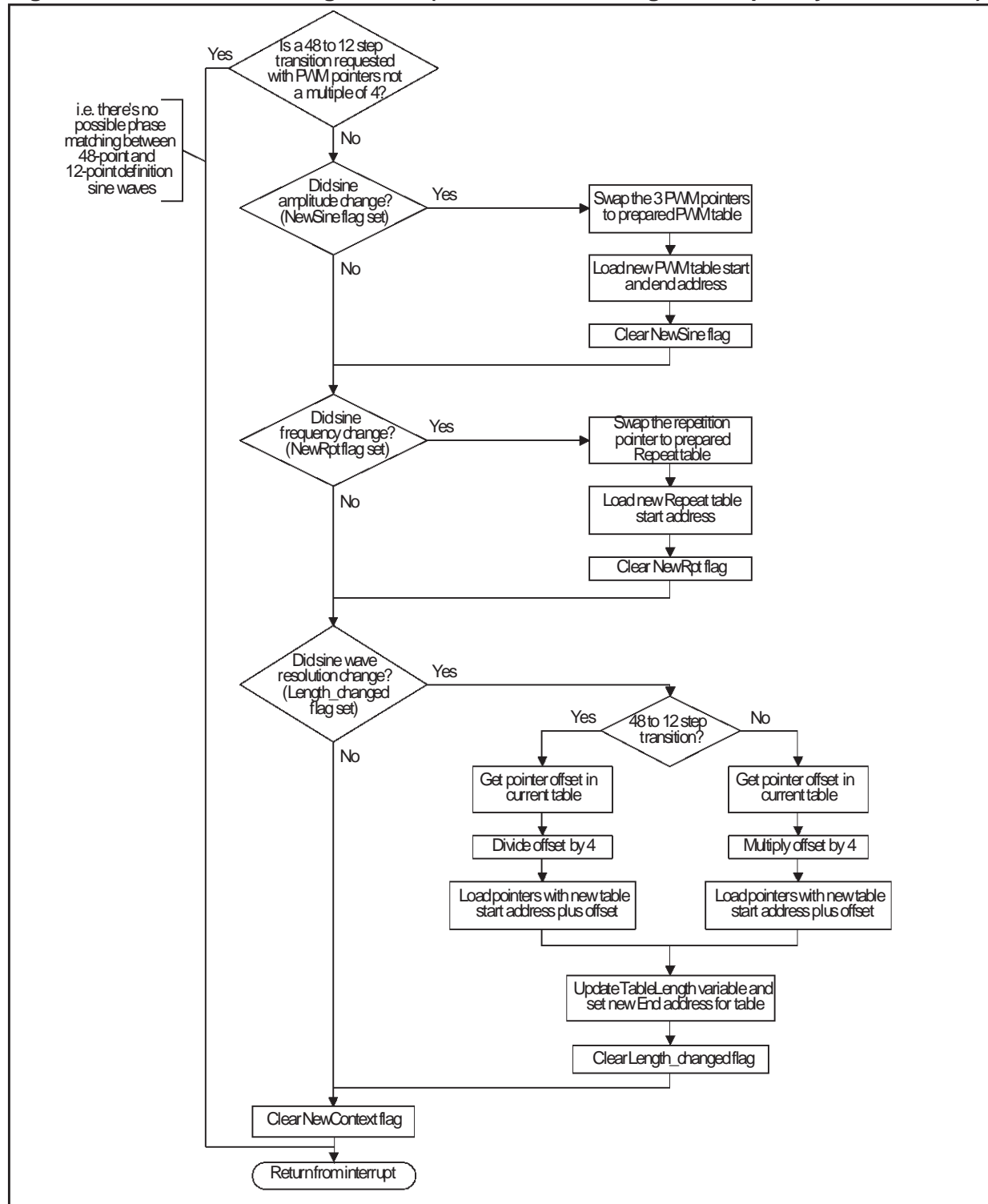


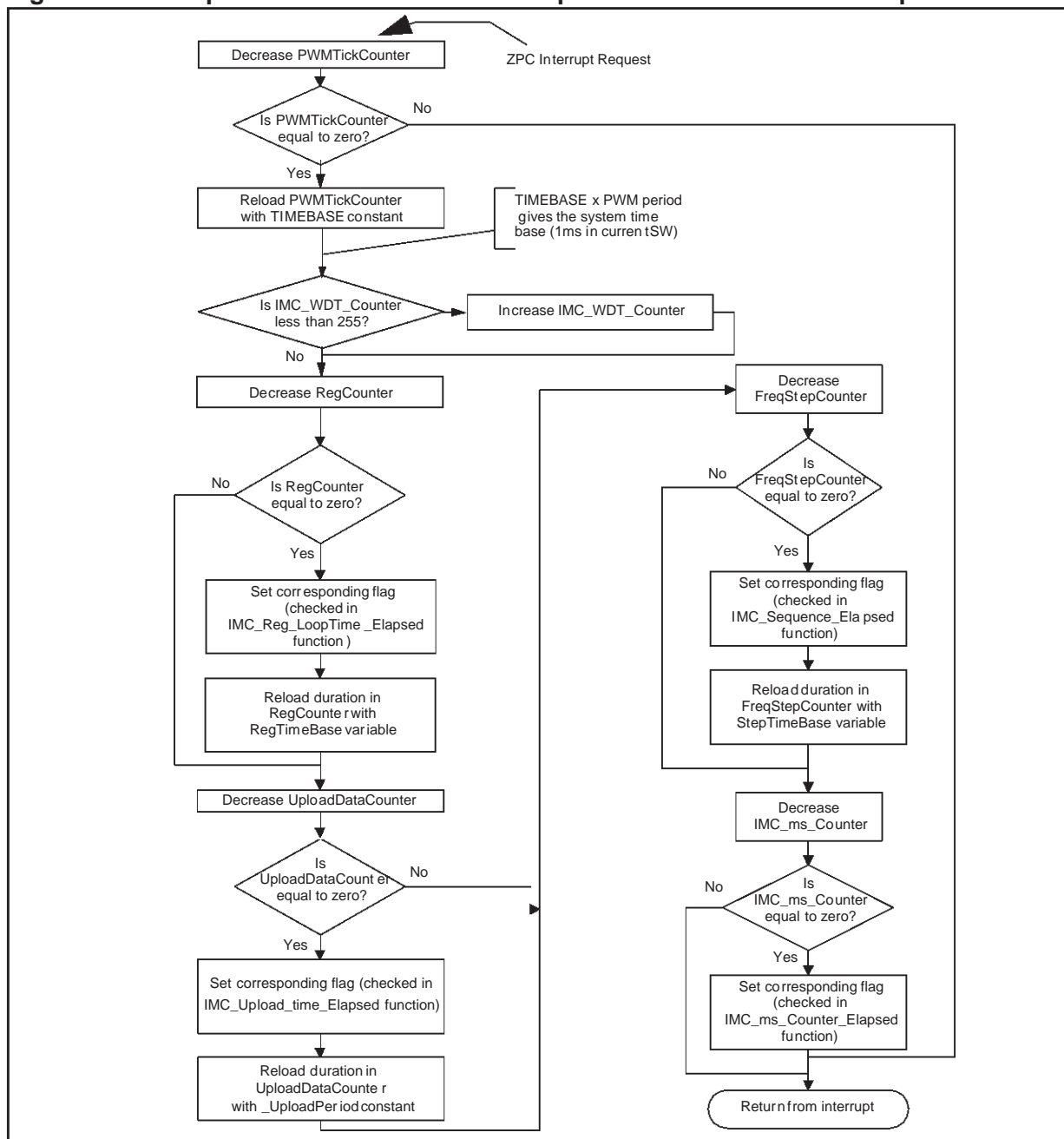
Figure 18. Context Switching in ADT (i.e. either the voltage or frequency are modified)



4.1.2 Zero of PWM Counter Interrupt (ZPC)

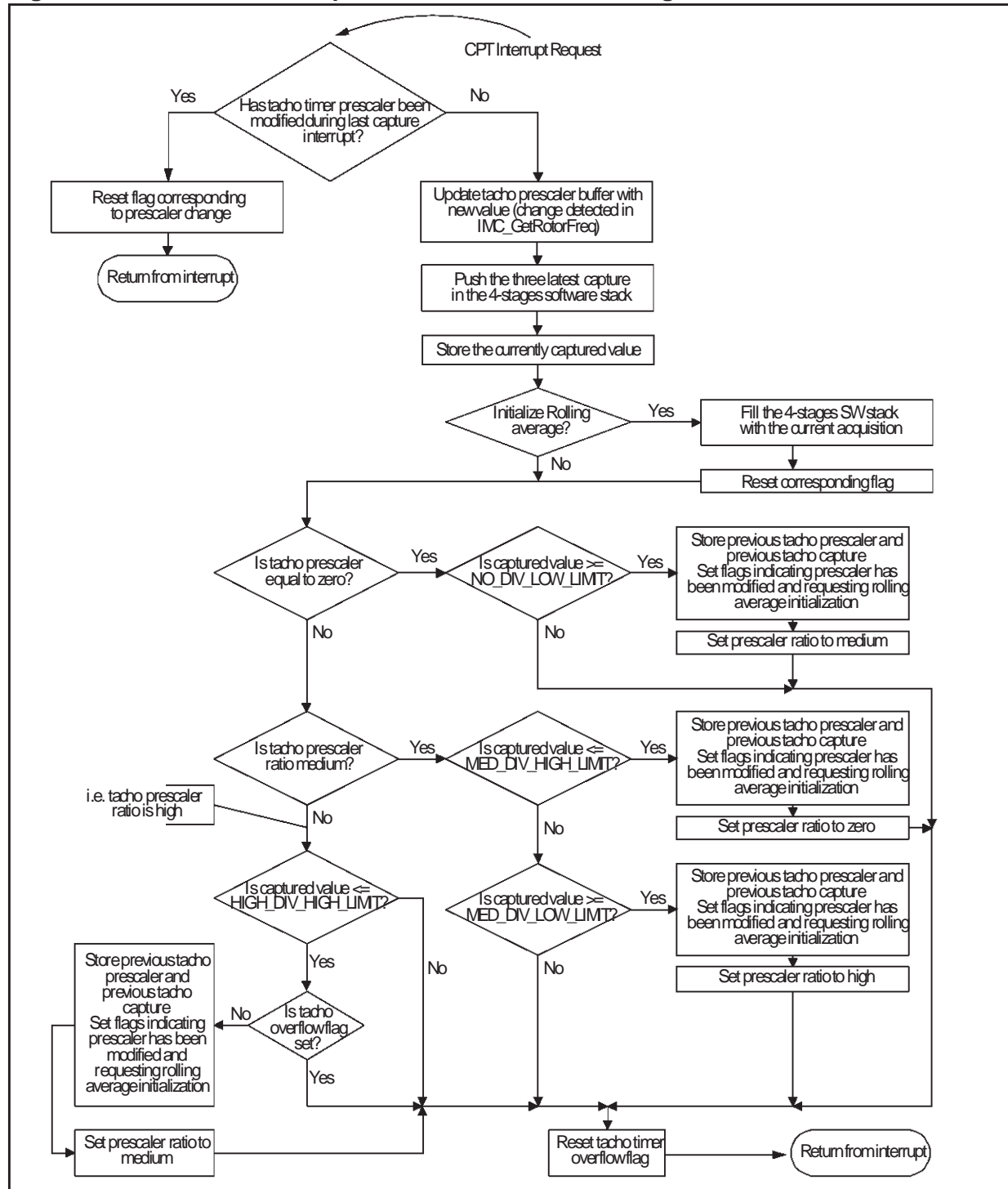
Note: Timebase is implemented with the ADT_IT routine in Version 2.0. Please refer to application note AN1277 for more details.

Figure 19. Example of Software Timebase implementation in ZPC Interrupt



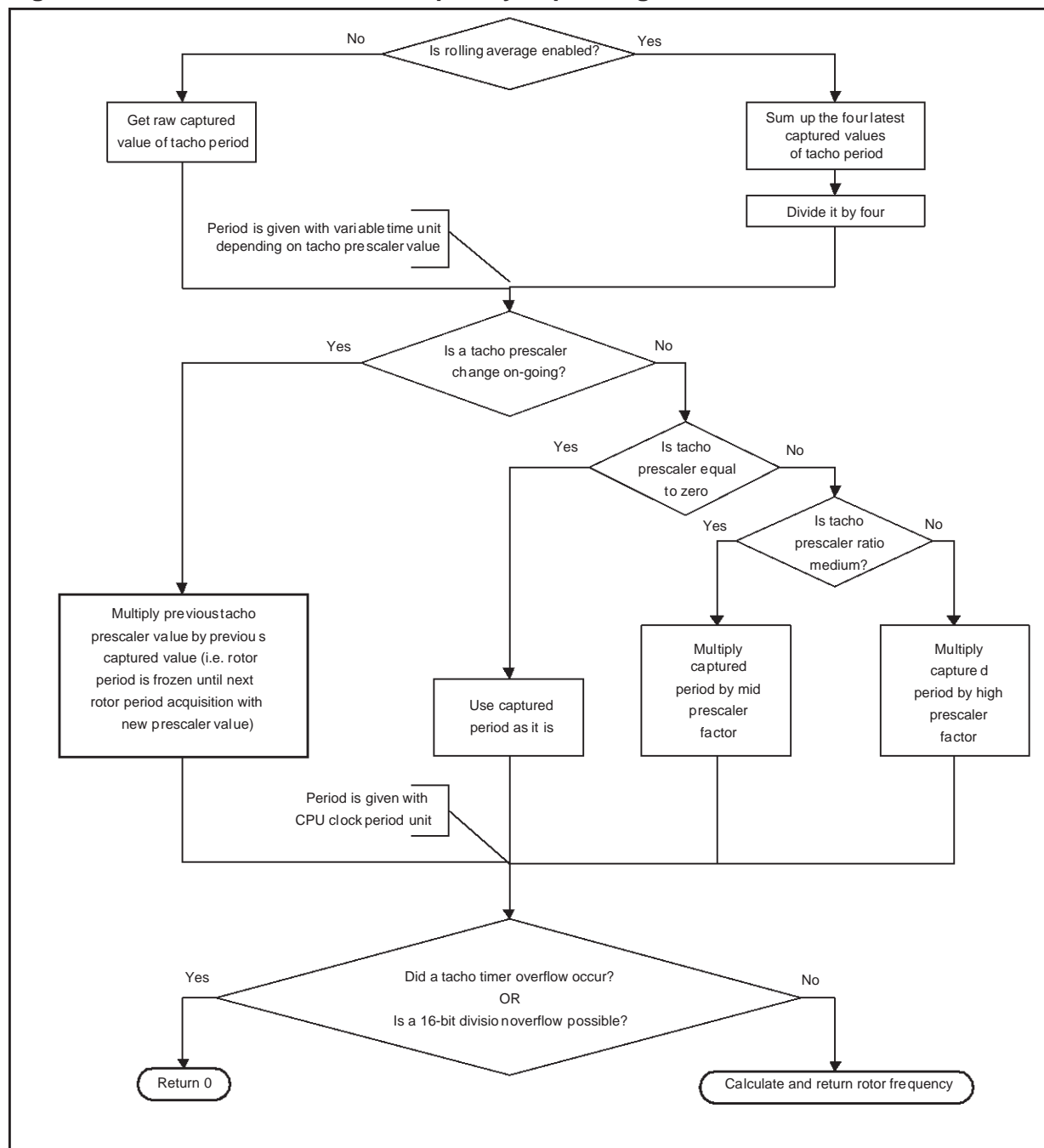
4.1.3 Tacho Capture Interrupt (CPT)

Figure 20. Tacho Period Acquisition with Auto-Prescaling



4.1.4 IMC_GetRotorFreq

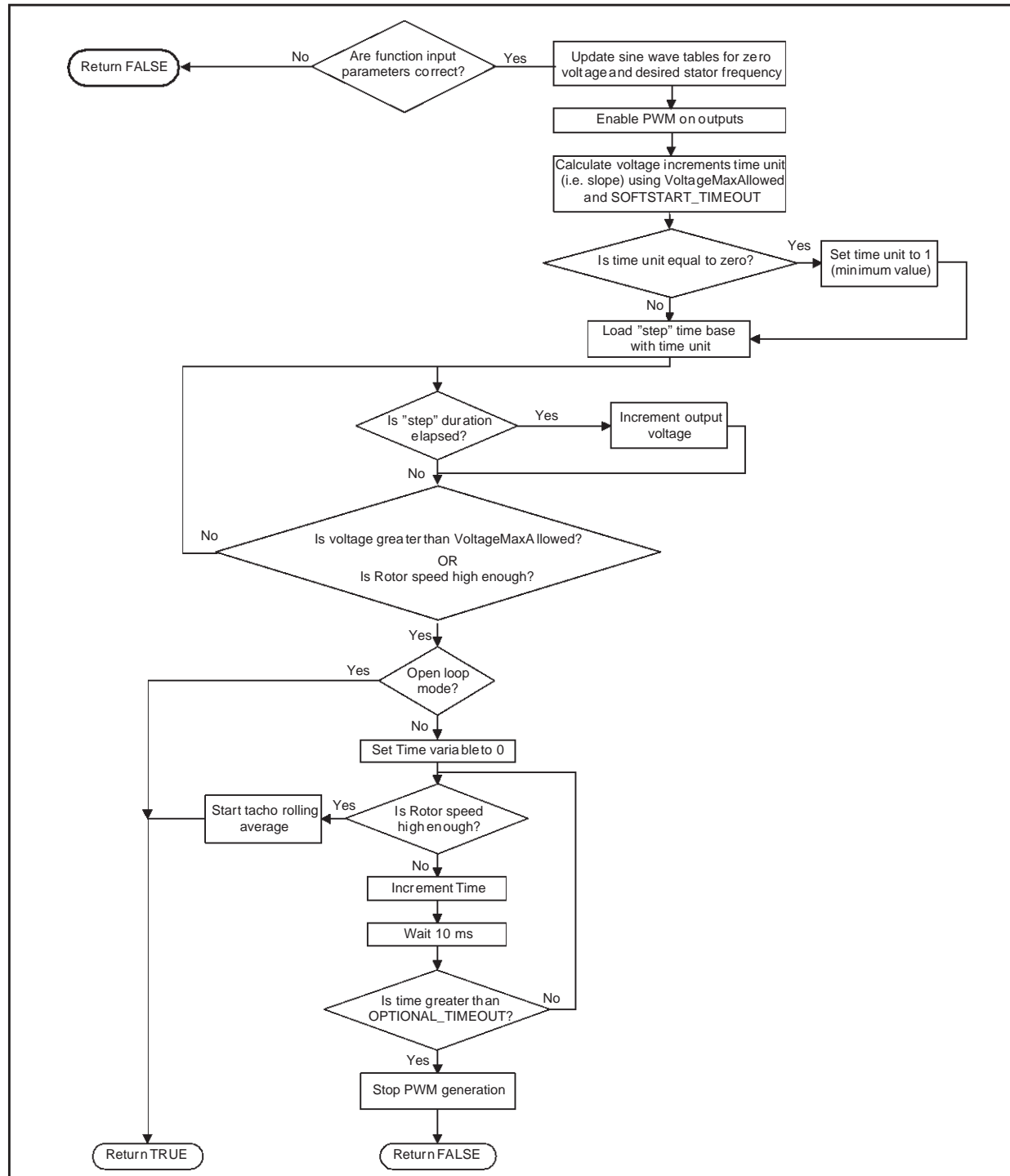
Figure 21. Calculation of Rotor Frequency depending on Tacho Prescaler Ratio



4.2 ACMOTOR MODULE FLOWCHARTS

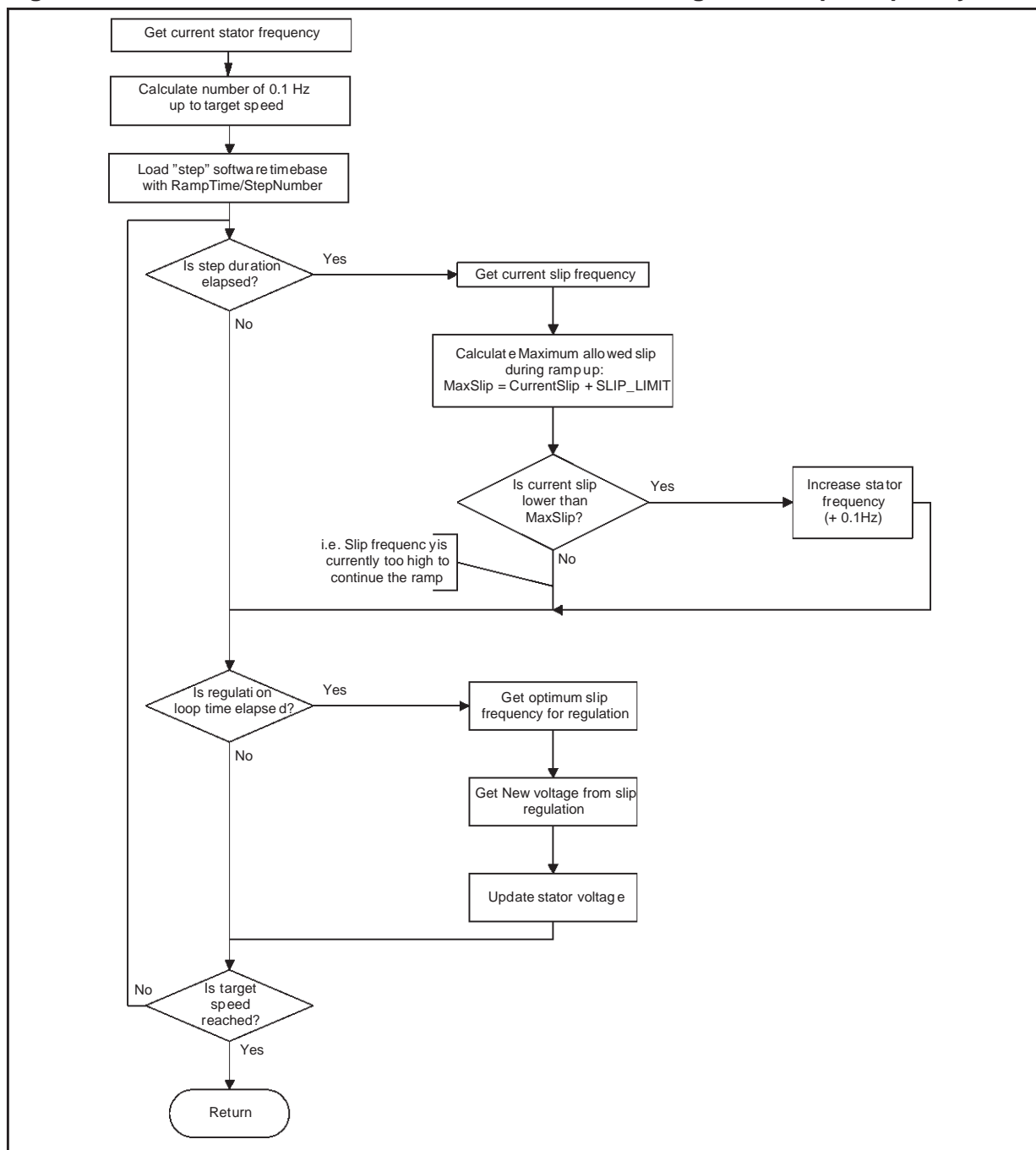
4.2.1 ACM_SoftStart

Figure 22. Starting Procedure limiting the Inrush Current and Start-up Torque



4.2.2 ACM_RampUp

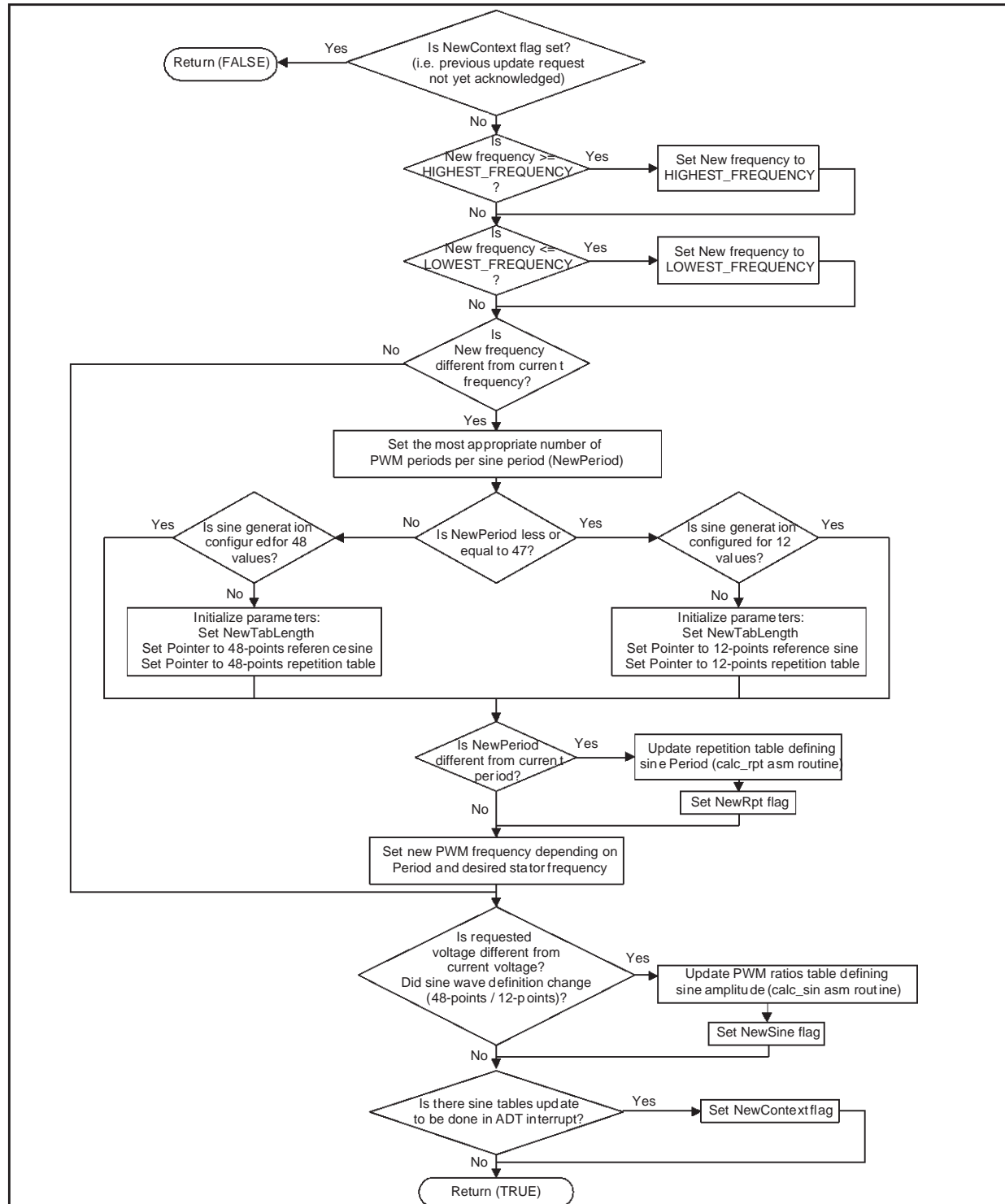
Figure 23. Motor Acceleration with Continuous Monitoring of the Slip Frequency



4.2.3 ACM_Update_Sine_Tables

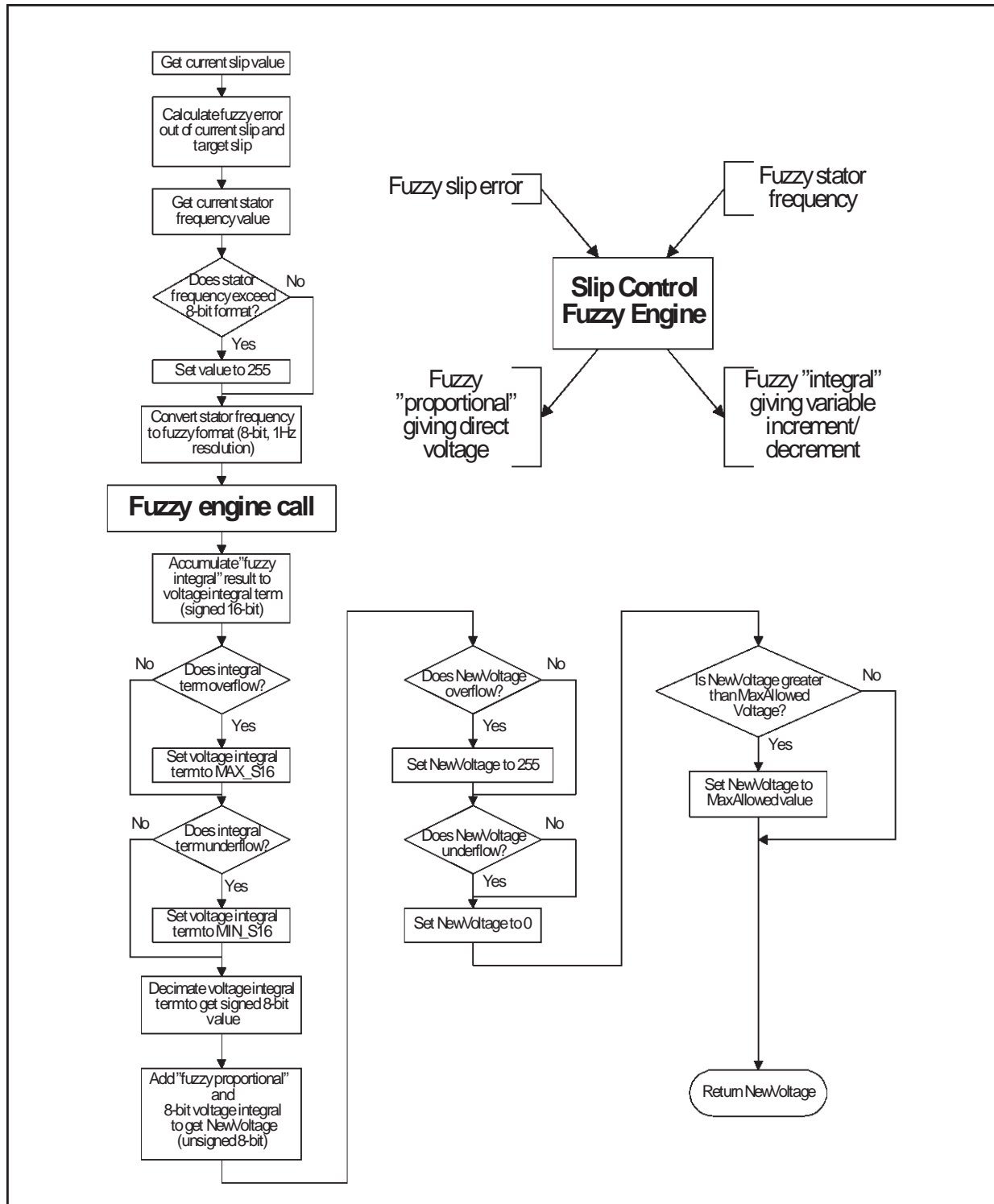
Note: Due to important changes in PWM sine wave generation methods, this flowchart is now obsolete for Version 2.0. Please refer to application note AN1277 for more details.

Figure 24. Output Sine Wave Parameter Modification (Voltage, Frequency)



4.2.4 ACM_SlipRegulation

Figure 25. Regulation Routine using a Fuzzy Engine



4.3 TOOL OPTION SUMMARY

Further details can be found in the corresponding literature (tool user manuals).

4.3.1 Linker

It is directly invoked (rather than via the GCC9). The following options are used to modify the behaviour of the linker program (LD9).

-I

During the link phase, the GCC9 (default) ensures that the initial values of initialized variables are copied to the end of the .text segment. From here (normally, in non-volatile memory) they will be recopied at run-time into the .data segment (in read/write memory) so that these values may be used and modified by the program. If the linker is directly invoked for linking (instead of via the GCC9) by default the LD9 linker does not copy the initial values to the .text segment. This is only done if option -I is specified.

In order for the .data section values to be copied to the .text area, they should be linked with the GCC9 (default) or the LD9 with option -I. If the initial values are not to be copied to the end of the .text segment, the GCC9 may be invoked with option -nol, or the LD9 in default mode (i.e. LD9 called without option -I).

When option -I is active, the LD9 will define the `_text_end` global symbol as the final address of the initialized variables stored within the .text section (i.e. at the end of the .text section after the initial content of the .data section);

-mmu

When the linker is directly invoked, this is not the default option and it must be specified when targeting ST9+ architecture. However, the GCC9 invokes the LD9 linker with the -mmu option by default. This generates an ST9+ application containing 64Kbyte memory code segments. Instructions specific to ST9+ MCUs requiring special relocations, new assembler operators and correct DPR translation are only possible under this option. The -mmu option also instructs the linker to use 22-bit physical addresses when generating a map listing, as well as to produce a table of DPR register assignments.

-m

This option writes the link map to the <filename>.map output file. When -o is not specified, the the link map is written to the a.map file by default.

-v

Verbose mode. This option is used to display the version number of LD9, as well as the files and libraries as they are processed, names of temporary open files, etc.

4.3.2 Assembler

It is invoked via the GCC9. The following options are passed to modify behaviour of the assembler program (GAS9).

-g

The **-g** option is used to produce debugging information that will be used by the GAS9 and the LD9 to allow source level debugging by the WGDB9. This option generates the stabs, stabd and stabn directives in the generated file.

It is important to bear in mind that the **-tr9** option is required to compile assembly programs that include TR9 files (*.inc) or to compile C programs that contain asm directives involving the TR9.

-tr9

By default, the compiler generates code for the GAS9 assembly language; TR9 is not invoked unless option **-tr9** is used. With this option, the GCC9 will invoke the CPP9 + CC9 (with the **-mtr9** option) + TR9 + GAS9 + LD9. Note that the behaviour of the compiler has been changed in versions 4.3 and higher -- the TR9 language treats all ST9+ instruction mnemonics as reserved words. In previous versions, this prevented the user from defining functions named **add**, **div** or **and**. This rendered the compiler non-ASCII compliant and difficult to use, especially if calling the library routine **div**, delivered with the ST9+ toolchain.

A number of options have been added to allow parameters to be passed selectively from the GCC9 to the TR9, GAS9 and LD9. These are respectively:

- **Wt**, for the macro-expander;
- **Wa**, for the assembler; and **WI**, for the linker-loader:
- GCC9 with **-Wt**, allows the passing of the string which follows the comma to TR9.

Note that **-tr9** must be invoked for this to happen (i.e. the **-tr9 -Wt,-c** command string will send the **-c** parameter to the TR9);

- GCC9 with the option **-Wa**, will pass the string which follows the comma to the assembler GAS9 (i.e. **-Wa,-ahld** to pass option **-ahld** to GAS9);
- GCC9 with **-WI**, allows the passing of the string which follows the comma to the linker (i.e. **-WI,-m**) to pass the **-m** parameter to the LD9).

4.3.3 C Compiler (GCC9)

The following options are passed to modify the behaviour of the C compiler (GCC9).

-samepd

The GCC9 **-samepd** option (or the CC9 **-mnopd** option) is used to generate ST9 code where program and data use only a single 64Kbyte space.

The **-samepd** option places all constants and const-type variables in the **.text** section.

-g

This option generates source level debugging information (this information is present in the generated file, using specific assembly directives, and will be used by the WGDB9+ during the debug phase). This option has no effect on the code generated.

-c

The **-c** option is used to stop after having invoked the assembler (GAS9). This option is also used to compile, macro-assemble or assemble the source files, but not to link them. Object files are produced with names formed by replacing the existing suffix with **.o**.

-O

Optimize. When the optimization option is selected, the compilation time is longer and the host will require more memory.

Without the **'-O'** option, the compiler will reduce compilation costs and ensure that the debugging will produce the expected results. Statements are independent, and the program may be stopped using a breakpoint between statements. A new value can be assigned to any variable, or the program counter set to any other statement in the function, with exactly the same results as expected from the source code. Only variables that are explicitly declared as register variables are allocated to registers.

When **'-O'** is invoked, the compiler will reduce code size and minimize the execution time of the generated code.

-mlink

This option generates shorter prologues and epilogues. The compiler is requested to use either **link** or **unlink** instructions in the prologue and epilogue. When the **-mparmusp** option is also active, the compiler will use **linku** and **unlinku**. By default, the compiler generates prologue and epilogue functions that are recognisable by ST9+ devices, and will adjust the system or user stack by using **pushw**, **ldw**, and **subw**.

When the **-mlink** option is selected, a **GCC9_LINK** macro is predefined by the C processor to flag that the new version of prologue and epilogue are in use.

-fomit-frame-pointer

The frame pointer should only be maintained in a register if it is used by functions. This eliminates unnecessary frame pointer save, setup and restore instructions, an extra register in many functions is also freed. The option decreases code size, but on the other hand, it sometimes makes debugging difficult.

-mparmusp

The **-mparmusp** option will use the user stack pointer (RR236) to pass function parameters and push register values. By default, the system stack pointer (RR238) is used.

If the **-mparmusp** option is invoked, the two stacks must be initialised in the C start-up file.

The ST9+ is able to use of 2 stacks: the system stack, where return addresses from calls and interrupts are stored, and the user stack. Variables in either stack may be pushed, popped or accessed.

The compiler uses only one stack, by default, the system stack. In this case, return addresses from calls are stored in this stack, as well as parameters and local variables. It is also used for internal temporary storage.

The **-mparmusp** option allows the user stack to be used. In this case, return addresses are still stored in the system stack, but parameters, local variables and temporary internal data are stored in the user stack. When this option is invoked, the GCC9 generates the predefined `GCC9_PARMUSP` macro for the preprocessor.

The algorithm used for parameter passing remains the same, except that the parameters are now are stored in the user stack, instead of the system stack.

Caution: The ST9+ is able to map stacks in either the register file or in memory. If only the system stack can be used, it **MUST** be mapped to memory. If both stacks are employed, the system stack may be mapped to the register file, but the user stack **MUST** be mapped to memory. The interest of using both stacks is that the system stack can be mapped in the register file, which decreasing the size occupied by the user stack in memory. However, this simultaneously reduces the size of the register file that remains available to the program.

Note: Library names that end with 'ureg9' must be used with the '**-mparmusp**' option. (The first parameters use registers rr0 and rr2, and the remaining ones are in the user stack)

-Wa,ahdl

If the GCC9 is used with the **-Wa** option, the string that follows the comma will be passed to the GAS9 assembler (i.e. **-Wa,-ahld** to pass option **-ahld** to GAS9); i.e. the **-a** option is used to request the generation of the listing file that includes high-level lines, assembly lines and a symbols list. This option should be followed by modifiers that specify which information should be included in the listing file. The modifiers must be concatenated with this option (i.e. "**-ahld**").

Modifiers:

- "l" is used to request the generated assembly code listing, the relative offset of the instruction from the beginning of the current section and the hexadecimal dump of the instruction. The use of this modifier is recommended.
- "h" is used to request the generation of source lines interleaved with the generated code. The use of this modifier is recommended.
- "d" is used to suppress the source-level directives included in the intermediate assembly file by the compiler or the macro-expander. These directives are generated by the source-level debugging option ('-g'). They are used by the GAS9, but they make the listing file difficult to read. The use of this modifier is recommended.

-Wall

This option specifies the maximum warning level.

4.4 CREATE YOUR OWN FTC8.L FILE

This file is required if certain tool options contained in the makefile must be modified. The procedure used to create a ftc8.l file is described below:

1. Modify the stmake8.bat batch file (described below) to include the new options.
2. Copy the "ftlibc.h" file (located in \fuzzyTECH 5.2\Runtime\C\include) and the stmake8.bat utility into the \fuzzyTECH 5.2\Runtime\C\Lib\Src\MCU folder.
3. Run the stmake8.bat utility in this last directory, using the toolchain DOS session. It will create a lib8 folder with object files and the ftc8.l fuzzy kernel library.
4. Copy the ftc8.l file into the fuzzylib directory of the working folder (Ex: c:\code\st92141).

Note: The same utility exists for using 16-bit internal variables in the fuzzy kernel. The identical procedure is used (stmake16.bat will replace stmake8.bat).

```
@echo off
echo *****
echo ***** STMAKE8.BAT: fuzzyTECH Runtime Kernel FTC8.LIB      April 2000
echo *****
echo ***** Step 1: Create a lib8 directory
echo ***** Step 2: Compile all C-Sources to object files
echo ***** Step 3: Move all object files to a library
echo *****
echo ***** Note:  The compiler command line switches used in this batch file
echo *****          for building the library is set for current ST92141lib SW.
echo *****          It uses 8-bit variables for fuzzy computations.
echo *****
echo *****
echo *****

mkdir lib8

gcc9 -samepd -g -c -O -mlink -fomit-frame-pointer -mparmusp -Wall -DFTLIBC8
hcom.c  -o lib8\hcom.o
gcc9 -samepd -g -c -O -mlink -fomit-frame-pointer -mparmusp -Wall -DFTLIBC8
hflms.c  -o lib8\hflms.o
gcc9 -samepd -g -c -O -mlink -fomit-frame-pointer -mparmusp -Wall -DFTLIBC8
hflmss.c -o lib8\hflmss.o
gcc9 -samepd -g -c -O -mlink -fomit-frame-pointer -mparmusp -Wall -DFTLIBC8
hmax.c  -o lib8\hmax.o
gcc9 -samepd -g -c -O -mlink -fomit-frame-pointer -mparmusp -Wall -DFTLIBC8
hmaxprod.c -o lib8\hmaxprod.o
gcc9 -samepd -g -c -O -mlink -fomit-frame-pointer -mparmusp -Wall -DFTLIBC8
hmin.c  -o lib8\hmin.o
gcc9 -samepd -g -c -O -mlink -fomit-frame-pointer -mparmusp -Wall -DFTLIBC8
```

Getting Started with the ST92141 Software Library Version 1.0

```
hminprod.c -o lib8\hminprod.o
gcc9 -samepd -g -c -O -mlink -fomit-frame-pointer -mparmusp -Wall -DFTLIBC8
hmom.c -o lib8\hmom.o
gcc9 -samepd -g -c -O -mlink -fomit-frame-pointer -mparmusp -Wall -DFTLIBC8
hpubvars.c -o lib8\hpubvars.o

cd lib8

ar9 qv ftc8.1 hcom.o hflms.o hflmss.o hmax.o hmaxprod.o hmin.o hminprod.o hmom.o
hpubvars.o

ar9 s ftc8.1

cd ..
```

THE PRESENT NOTE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH A NOTE AND/OR THE USE MADE BY CUSTOMERS OF THE INFORMATION CONTAINED HEREIN IN CONNEXION WITH THEIR PRODUCTS."

Information furnished is believed to be accurate and reliable. However, STMicroelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of STMicroelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. STMicroelectronics products are not authorized for use as critical components in life support devices or systems without the express written approval of STMicroelectronics.

The ST logo is a registered trademark of STMicroelectronics

©2000 STMicroelectronics - All Rights Reserved.

Purchase of I²C Components by STMicroelectronics conveys a license under the Philips I²C Patent. Rights to use these components in an I²C system is granted provided that the system conforms to the I²C Standard Specification as defined by Philips.

STMicroelectronics Group of Companies

Australia - Brazil - China - Finland - France - Germany - Hong Kong - India - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain
Sweden - Switzerland - United Kingdom - U.S.A.

<http://www.st.com>