
Bermain-main dengan Mikrokontroler ARM Cortex-M3 STM32F207

Usman

Penerbit

<https://karedox.com>

by Kang U-2Man (u_2man@yahoo.co.id)

KATA PENGANTAR

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

Sudah 8 tahun sejak saya pertama kalai menuliskan huruf pertama untuk buku ini. Awalnya sangat bersemangat untuk segera menyelesaikan buku ini. Di setiap ada kesempatan pasti akan meluangkan waktu untuk menyelesaiannya. Pernah dulu waktu di busway lagi mengetik menggunakan aplikasi pengolah kata di HP, ada yang nanya, Bapak lagi nulis buku ya? Saya jawab iya. Namun seiring waktu, karena kesibukan saya dan juga rasa malas (ini yang lebih dominan), proyek ini terbengkalai, walaupun sejauh ini sudah menulis lebih dari 300 halaman.

Rencana awalnya sih dikasih ke penerbit, dicetak kemudian dapat royalti. Namun mengingat hal di atas tadi, akhirnya saya memutuskan mempublikasikan sendiri buku ini dalam bentuk e-book (format pdf) dan pastinya saya akan menggratiskan buku ini alias tidak akan meminta royalti, tapi ya kalau ada yang mau donasi sih tidak akan saya tolak..hehehe.

Saya menyadari bahwa buku ini jauuuuuuuuh dari sempurna, selesai aja tidak. Tetapi mudah-mudahan buku ini bisa memberikan pengetahuan dasar tentang bagaimana ngoprek menggunakan mikrokontroler STM32 (khususnya STM32F207). Setelah membaca buku ini, minimal pembaca bisa membuat program blink LED dengan STM32. Buku ini mengkhususkan STM32F207, namun contoh program yang ada di buku ini akan sangat mudah untuk di-porting ke tipe STM32 yang lain atau bahkan ke mikrokontroler buatan pabrikan selain ST Micro (TI, NXP dll).

Banyak bab dibuku ini yang belum selesai atau bahkan belum ada isinya. Sebagai contoh Bab 3 yang menerangkan tentang STM32F207. Untuk melengkapinya pembaca diharapkan untuk mengacu kepada *Reference Manual STM32F207*, seperti yang tercantum di daftar pustaka. Bab 10 juga rencananya akan membahas bagaimana menampilkan file jpg/jpeg

sedangkan Bab 11 – Bab 17 belum ada isinya sama sekali. Insya Allah mudah-mudahan saya diberi kesempatan untuk melengkapinya.

Mengingat banyaknya kekurangan yang ada dibuku ini, saya akan dengan senang hati menerima saran dan kritik dari pembaca. Atau bahkan kalau ada yang mau melengkapi bab atau sub-bab yang masing sangat tidak lengkap itu, tentunya akan sangat berterima kasih. Seperti software open source, semakin banyak yang berkontribusi akan semakin baik.

Buku ini, tentu saja, karena sudah saya nyatakan bebas royalti, maka siapa saja boleh memiliki, meng-copy, menyebarkan bahkan mencetaknya tanpa perlu meminta izin kepada saya. Hal yang tidak diperbolehkan adalah menghilangkan kredit saya sebagai penulis, misal dengan meng-copy atau menulis ulang buku ini dan mengganti identitas penulis dengan penulis lain. Mengkomersilkan buku ini untuk keuntungan pribadi juga tidak diperbolehkan. Urusannya pasti akan diselesaikan di akhirat...hehe.

Terakhir, semoga buku yang tidak sempurna ini bisa memberikan manfaat amal jariyah kepada saya dan manfaat ilmu pengetahuan kepada siapa saja yang membacanya. Aamiin.

Purwakarta, 27 Desember 2021

Kang Usman (karedox.com)

DAFTAR ISI

KATA PENGANTAR	III
DAFTAR ISI	V
BAB 1 PENDAHULUAN	1
BAB 2 ARSITEKTUR ARM CORTEX-M3	7
2.1 MODEL PROGRAMMER	10
2.1.1 MODE OPERASI	10
2.1.2 REGISTER	11
2.2 ARSITEKTUR BUS	16
2.3 PIPELINE	18
2.4 MODEL MEMORI	20
2.4.1 PEMETAAN MEMORI	20
2.4.2 AKSES MEMORI	22
2.4.3 OPERASI BIT BAND	24
2.4.4 AKSES MEMORI TAK RATA	28
2.5 MODEL EKSEPSI	29
2.5.1 HARD FAULT	31
2.5.2 MEMMANAGE FAULT	32
2.5.3 BUS FAULT	34
2.5.4 USAGE FAULT	35
2.5.5 SVC, PENDSV DAN SYSTICK	36
2.5.6 TABEL VEKTOR	39
2.6 KENDALI INTERUPSI	42
2.6.1 REGISTER-REGISTER NVIC	43
2.6.2 PRIORITAS INTERUPSI	47
2.6.3 KARAKTERISTIK INTERUPSI	50
2.6.4 SETING INTERUPSI	54
2.7 UNIT PROTEKSI MEMORI	57

2.7.1	REGISTER-REGISTER MPU	58
2.7.2	PENGGUNAAN MPU	62
2.8	FITUR LAIN CORTEX-M3	63
2.8.1	TIMER SYSTICK	63
2.8.2	MANAJEMEN DAYA	66
2.8.3	KOMUNIKASI MULTIPROSESOR	67
2.8.4	KENDALI RESET	68
2.9	SISTEM DEBUG	69
2.10	SET INSTRUKSI ARM DAN THUMB-2	73

BAB 3 MIKROKONTROLER STM32F207

3.1	SISTEM ARSITEKTUR DAN MEMORI	79
3.1.1	SISTEM BUS MATRIK	79
3.1.2	PEMETAAN MEMORI	81
3.1.3	MEMORI FLASH	83
3.1.4	KONFIGURASI BOOT	84
3.2	GPIO	86
3.2.1	FITUR-FITUR	86
3.2.2	PENGATURAN GPIO	87
3.3	TIMER	95
3.3.1	FITUR UTAMA	98
3.3.2	MODE KERJA TIMER	102
3.4	ANALOG TO DIGITAL CONVERTER	117
3.4.1	FITUR UTAMA	117
3.4.2	MODE KERJA	121
3.5	DIGITAL TO ANALOG CONVERTER	133
3.5.1.	FITUR UTAMA	134
3.5.2.	MODE KERJA	136
3.6	KOMUNIKASI SERIAL UART/USART	141
3.6.1	FITUR UTAMA	142
3.6.2	MODE KERJA	143
3.7	SPI	154
3.7.1	FUNGSI KERJA SPI	156
3.7.2	FUNGSI KERJA I2S	162
3.8	I2C	167
3.8.1	FUNGSI KERJA I2C	168
3.8.2	FUNGSI SMBUS	171
3.9	CAN	173

3.9.1	FUNGSI KERJA CAN	174
BAB 4 SISTEM MINIMUM STM32F207		177
4.1	CATU DAYA	177
4.1.1	SKEMA CATU DAYA STM32F207	178
4.1.2	REKOMENDASI RANCANGAN	179
4.2	SISTEM RESET	181
4.3	CLOCK	183
4.4	KONFIGURASI BOOT	185
4.5	SISTEM DEBUG	186
BAB 5 ALAT PENGEMBANGAN		189
5.1	PROBE DEBUG	189
5.2	IDE	190
5.3	SOFTWARE PENDUKUNG	196
5.3.1	STM32CUBEMX	196
5.3.2	ALAT PEMROGRAM FLASH	197
5.3.3	PROGRAM TERMINAL	198
5.3.4	PROGRAM PENGANALISA PROTOKOL	200
5.4	HAL, CMSIS DAN MIDDLEWARE	201
BAB 6 BERMAIN-MAIN DENGAN PERIPERAL DASAR		205
6.1	BERMAIN-MAIN DENGAN GPIO	205
6.1.1	GPIO SEBAGAI OUTPUT	206
6.1.2	GPIO SEBAGAI INPUT	227
6.1.3	GPIO SEBAGAI SUMBER INTERUPSI	229
6.2	BERMAIN-MAIN DENGAN UART	234
6.2.1	KIRIM DAN TERIMA DATA SERIAL	235
6.2.2	FUNGSI PRINTF KE PORT SERIAL	241
6.2.3	TEKNIK PARSING DATA	245
6.3	BERMAIN-MAIN DENGAN TIMER	254
6.3.1	TIMER SEBAGAI SUMBER PEWAKTUAN	254
6.3.2	TIMER SEBAGAI PEMBANGKIT SINYAL PWM	259
6.3.3	TIMER SEBAGAI PENGUKUR SINYAL	263
6.4	BERMAIN-MAIN DENGAN I2C	266

BAB 7 BERMAIN-MAIN DENGAN ADC/DAC	273
7.1 BERMAIN-MAIN DENGAN ADC	273
7.1.1 PENGUKURAN TEGANGAN DC	274
7.1.2 PENGATURAN PWM DENGAN ADC	277
7.2 BERMAIN-MAIN DENGAN DAC	279
7.2.1 PEMBANGKITAN TEGANGAN DC DENGAN DAC	280
7.2.2 PEMBANGKITAN SINYAL SEGITIGA DAN NOISE	283
BAB 8 BERMAIN-MAIN DENGAN DMA	287
8.1 UART DAN I2C DENGAN DMA	288
8.1.1 TRANSFER DATA UART DENGAN DMA	288
8.1.2 TRANSFER DATA I2C DENGAN DMA	294
8.2 GENERATOR SINYAL DENGAN DAC	295
8.2.1 DMA MODE NORMAL	298
8.2.2 DMA MODE CIRCULAR	302
BAB 9 BERMAIN-MAIN DENGAN KARTU MIKRO SD DAN FATFS	305
9.1 MIKRO SD	306
9.2 SISTEM FILE	309
9.3 FATFS	311
9.4 BERMAIN-MAIN DENGAN FATFS	315
9.4.1 OPERASI BACA/TULIS FILE	315
9.4.2 JAM DAN TANGGAL AKSES FILE	321
9.4.3 MIKRO SD SEBAGAI PENGGANTI EEPROM	326
BAB 10 BERMAIN-MAIN DENGAN LCD TFT + TOUCH PANEL	331
10.1 PRINSIP KERJA LCD TFT	331
10.2 PRINSIP KERJA LAYAR SENTUH	334
10.3 ANTARMUKA LCD DENGAN STM32F207	335
10.4.1 ANTARMUKA	335
10.3.1 PENGALAMATAN	338
10.3.2 PEWAKTUAN	340
10.4.2 LCD - HELLO WORLD	341
10.4.3 MENAMPILKAN FILE BMP	341
	356

BAB 11 BERMAIN-MAIN DENGAN DEKODER MP3 + MIDI	363
BAB 12 BERMAIN-MAIN DENGAN FILE VIDEO	365
BAB 13 BERMAIN-MAIN DENGAN KAMERA DIGITAL	367
BAB 14 BERMAIN-MAIN DENGAN USB	369
BAB 15 BERMAIN-MAIN DENGAN ETHERNET	371
BAB 16 BERMAIN-MAIN DENGAN RTOS	373
BAB 17 BERMAIN-MAIN DENGAN IAP	375
DAFTAR PUSTAKA	377

by Kang U-2Man (u_2man@yahoo.co.id)

Bab 1

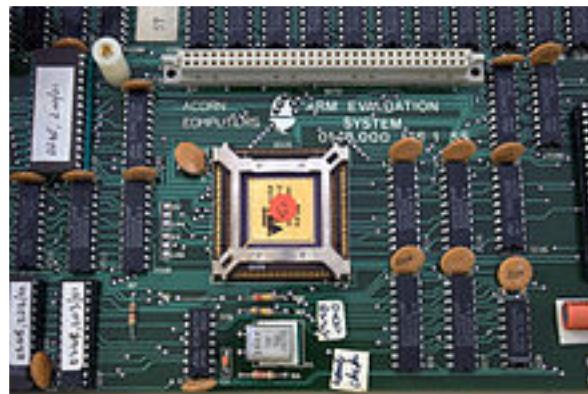
PENDAHULUAN

Mikrokontroler ARM pertama kali dikembangkan oleh sebuah perusahaan computer Inggris, Acorn Computer pada tahun 1980-an, yang dikembangkan untuk komputer BBC Micro. Dan ARM awalnya merupakan singkatan dari Acorn RISC (*Reduced Instruction Set Computing*) Machine.

Pada awalnya Acorn mempertimbangkan untuk menggunakan prosesor 6500 buatan Motorola (sekarang Freescale) atau 32016 dari National Semiconductor, namun tidak memenuhi persyaratan untuk dipakai di program computer BBC Micro. Akhirnya Acorn memutuskan untuk merancang prosesor sendiri. Proyek ini dipimpin oleh Steve Furber dan Sophie Wilson. Wilson mengembangkan set instruksi, menulis sebuah simulasi dari prosesor yang akan dikembangkan dalam bahasa BBC Basic yang dijalankan menggunakan prosesor 6502. Proyek ARM secara resmi dimulai di bulan Oktober 1983. Acorn menggandeng VLSI Technology sebagai mitra dalam pengembangan. Perusahaan perancangan dan pembuatan IC inilah yang menyediakan ROM dan chip khusus buat Acorn.

VLSI pertama kali membuat silicon ARM pada 26 April 1985, dinamai sebagai ARM1. Dan pada tahun berikutnya ARM2 siap untuk diproduksi. ARM2 memiliki bus data 32 bit, alamat 26 bit dan 27 buah register 32 bit. Bus alamat dikembangkan menjadi 32 bit di ARM6, tetapi ukuran memori program tetap 64MB untuk menjaga kompatibilitas. ARM2 hanya memiliki 30 ribu transistor di dalamnya (bandingkan dengan 68000 Motorola yang memiliki 68 ribu). Hal ini dikarenakan ARM2 tidak dilengkapi dengan Microcode dan cache. Hal ini membuat

ARM2 lebih hemat daya, lebih baik dari pada prosesor Intel 80286. Versi prosesor ARM berikutnya, ARM3, telah dilengkapi dengan cache 4KB, untuk meningkatkan performa.



Gambar 1.1 Prosesor ARM1 untuk BBC Micro

Di akhir tahun 1980-an Apple Computer (Sekarang Apple) mulai bergabung untuk membuat core ARM yang baru. Pada saat itu Apple tengah mengembangkan sebuah *Personal Digital Assistant* (PDA) dengan nama "Newton". Apple tertarik dengan prosesor baru yang sedang dikembangkan tersebut karena konsumsi dayanya yang sangat rendah untuk dipakai di PDA-nya. Karena masalah *intellectual property* (IP) maka diputuskan untuk membuat sebuah perusahaan baru. Maka pada tahun 1990, tepatnya 27 November 1990, berdirilah Acorn RISC Machines Ltd, dan kemudian menjadi ARM Ltd di tahun 1998, dengan induk perusahaan ARM Holding plc. Singkatan ARM kemudian menjadi Advanced RISC Machines. Apple menempatkan modal sebesar £1.5 Juta, Acorn menempatkan 12 orang insinyurnya sedangkan VLSI Technology menyediakan peralatan untuk perancangan.

Sejak pertama kali dibuat, core ARM telah banyak digunakan di peralatan elektronik (*embedded system*), mulai dari mesin cuci, meteran listrik, smart phone sampai laptop. Di mulai dengan core versi ARMv4T dengan serinya ARM7TDI sampai dengan ARM Cortex seri A yang banyak digunakan di merk-merk smart phone terkenal. Pada tahun 2010 core ARM mendominasi 95% di pasar smart phone, 10% komputer mobil, 35% di TV digital dan set top box. Sayangnya belum ada komputer desktop dan server yang menggunakan prosesor ARM. Walaupun demikian dengan keluarnya sistem operasi Windows 8 yang berbasis ARM besutan

Microsoft, diperkirakan di 2015 23% PC di dunia akan menggunakan prosesor ARM.

Di situs resminya (www.arm.com), ARM mengklasifikasikan prosesor ARM klasik dan prosesor ARM Cortex. ARM klasik merupakan prosesor dengan arsitektur ARMv6 ke bawah. ARM klasik terdiri dari 3 core ARM7, ARM9 dan ARM11. Dengan ARM7TDMI merupakan prosesor 32 bit yang paling banyak diproduksi di pasaran.

ARM Cortex terdiri dari beberapa seri:

- ARM Cortex-A50: menggunakan arsitektur ARMv8-A. Cortex-A50 merupakan prosesor 32 bit dengan kemampuan 64 bit (Aarch64). Digunakan untuk mendukung sistem operasi (OS) 64 bit. Contohnya prosesor Apple A7 Cyclone yang digunakan di iPhone 5S.
- ARM Cortex-A (Application): berarsitektur ARMv7-A, digunakan untuk menjalankan platform OS, seperti Linux, Android, dengan kemampuan grafis dan multimedia. Digunakan di smartphone, tablet TV digital dan lain-lain. Prosesor yang termasuk seri ini adalah Cortex-A5, -A7, -A8, -A9, -A12 dan -A15.
- ARM Cortex-R (Real Time): berarsitektur ARMv7-R. merupakan core prosesor 32 bit untuk sistem embedded yang memerlukan respon yang real-time dan performa yang sangat tinggi. Cortex-R terdiri dari -R4, -R5 dan -R7.
- ARM Cortex-M (Microcontroller): merupakan prosesor 32 bit digunakan untuk aplikasi low end di mana biaya sangat diperhitungkan. Karena harganya yang cukup kompetitif, prosesor Cortex-M diramalkan akan menggantikan prosesor 8/16 bit. Seri Cortex-M meliputi Cortex-M0, Cortex-M0+ dan Cortex-M1, dengan arsitektur ARMv6-M, Cortex-M3 (ARMv7-M) dan Cortex-M4 (ARMv7E-M) dengan kemampuan floating point dan DSP (Digital Signal Processing).

Selain prosesor (CPU), ARM juga mengembangkan Graphic Processor Unit (GPU) dengan kode MALI yang mendukung fungsi kemampuan 3D, prosesor SecurCore untuk aplikasi yang membutuhkan tingkat keamanan yang tinggi, seperti perbankan, e-Government dan lain-lain.

Tabel berikut meringkas core ARM yang telah dikembangkan dari pertama sampai sekarang, baik yang dikembangkan sendiri oleh ARM atau oleh perusahaan lain. Perlu dibedakan antara versi arsitektur dengan

versi core, misalnya core ARM7 memiliki arsitektur ARMv3 (arsitektur versi 3).

Tabel 1.1 Daftar Core Prosesor ARM

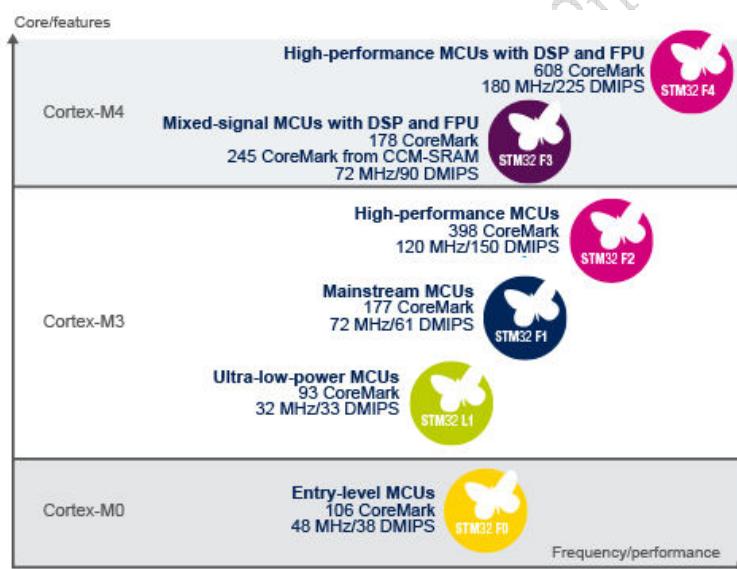
Arsitektur	Lebar Bit	Core Didesign Oleh ARM	Core Didesign oleh Perusahaan Lain	Profil Cortex
ARMv1	32/26	ARM1		
ARMv2	32/26	ARM2, ARM3	Amber	
ARMv3	32	ARM6, ARM7		
ARMv4	32	ARM8	StrongARM, FA526	
ARMv4T	32	ARM7TDMI, ARM9TDMI		
ARMv5	32	ARM7EJ, ARM9E, ARM10E	XScale, FA626TE, Feroceon, PJ1/Mohawk	
ARMv6	32	ARM11		
ARMv6-M	32	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1		Mikrokontroler
ARMv7-M	32	ARM Cortex-M3		Mikrokontroler
ARMv7E-M	32	ARM Cortex-M4		Mikrokontroler
ARMv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7		Real Time
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15	Krait, Scorpion, PJ4/Sheeva, Apple A6/A6X (Swift)	Application
ARMv8-A	64/32	ARM Cortex-A53, ARM Cortex-A57	X-Gene, Denver, Apple A7 (Cyclone)	Application
ARNv8-R	32	Belum diumumkan		Real Time

ARM Ltd bukanlah perusahaan manufaktur semikonduktor yang memproduksi IC/chip dari core prosesor yang mereka kembangkan. ARM adalah perusahaan yang menjual lisensi core dan arsitektur yang

mereka kembangkan kepada perusahaan-perusahaan semikonduktor, seperti Texas Instrument, ST Microelectronics, Atmel, Samsung, NXP dan lain-lain. Ada dua jenis lisensi yang dijual oleh ARM, yaitu lisensi core dan lisensi arsitektur.

Dengan lisensi core, para pabrikan semikonduktor akan menggunakan core ARM dan mengintegrasikan dengan berbagai peripheral yang akan membentuk System on Chip (SoC) sesuai dengan tujuan design dari perusahaan tersebut. Sedangkan lisensi arsitektur akan mengijinkan sebuah perusahaan untuk merancang core prosesor sendiri menggunakan instruksi-instruksi yang dimiliki oleh prosesor ARM, tentu saja core ini harus sesuai dengan standar yang dimiliki oleh core ARM.

ST (SGS-Thompson) Microelectronics, selanjutnya disebut ST, sebagai salah satu pemegang lisensi dari ARM, telah banyak memproduksi seri prosesor ARM, mulai dari ARM7 sampai ARM Cortex-A9 dan GPU Mali. Di kelas ARM7 dan ARM9 ST menghadirkan seri STR7 dan STR9, di seri Cortex-M ST memproduksi Cortex-M0, M3 dan M4, seperti terlihat di gambar 1.2.



Gambar 1.2 ARM Cortex-M yang Diproduksi oleh ST Microelectronics

Di seri ARM Cortex-M3 ada 3 keluarga STM32:

1. *Ultra-low MCU*, STM32L1, dengan kemampuan clock sampai 33 MHz.

2. *Mainstream MCU*, STM32F1, dengan kemampuan clock sampai 72MHz, dan
3. *High performance MCU*, dengan kemampuan clock sampai 120MHz.

STM32F207, yang akan dibahas di buku ini, merupakan mikrokontroler ARM Cortex-M3 yang masuk ke kategori High performance MCU. Mikrokontroler ini mempunyai memori flash kecepatan tinggi sampai 1Mbyte, SRAM (Static RAM) sampai 128KB, dengan frekuensi CPU sampai 120 MHz. Dilengkapi dengan peralatan internal standar seperti ADC 12 bit, DAC, RTC, 16 buah Timer, I2C, SPI, USART, dan CAN. Selain itu dilengkapi juga dengan *full speed* USB (host dan device) yang mendukung juga USB OTG (On The Go), Ethernet 10/100, antarmuka kamer 8-14 bit, dan antarmuka untuk memori eksternal (Flash, SRAM, PSRAM, NOR dan NAND) yang disebut dengan Flexible Static Memory Controller (FSMC). Fitur lebih lengkap bisa dilihat di datasheet STM32F207.

Buku ini berusaha menjelaskan bagaimana membuat sebuah sistem elektronik atau sistem embedded (*embedded system*) berbasis mikrokontroler STM32F207 ini. Dimulai dengan bagaimana memanfaatkan berbagai perangkat internal yang dimiliki oleh STM32F207, seperti yang telah dijelaskan di atas, antarmuka dengan IC dekoder MP3, display TFT, ethernet, sensor, USB dan lain-lain. Walaupun hanya berupa contoh sederhana, tapi dengan pengembangan lebih lanjut bisa menjadi aplikasi canggih.

Contoh-contoh aplikasi di buku ini akan menggunakan bahasa C sebagai bahasa pemrograman. Oleh karena itu, pemahaman mengenai dasar-dasar pemrograman bahasa mutlak diperlukan. Software pengembangan (development tool) akan menggunakan MDK ARM dari Keil dan CoIDE dari Coocox.org. MDK ARM merupakan software berbayar, versi Litenya bisa di-download di www.keil.com dengan file hexa yang dihasilkan dibatasi sebesar 32KB. Sedangkan CoIDE merupakan software gratis, compiler yang digunakan juga gratis (ARM GCC).

Bab 2

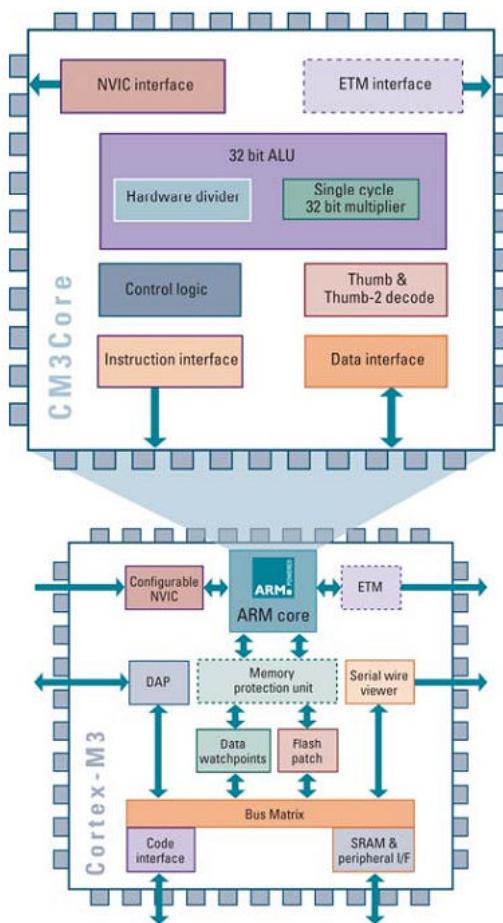
ARSITEKTUR ARM CORTEX-M3

Diperkenalkan tahun 2006, ARM Cortex-M3 merupakan prosesor 32 bit yang dirancang untuk aplikasi real-time dengan harga rendah. Cortex-M3 berbasis arsitektur ARMv7-M yang dirancang khusus untuk aplikasi sistem berbasis mikrokontroler, sistem otomotif, sistem kendali industri, dan jaringan nirkabel. Arsitektur Harvard yang digunakannya, di mana ada pemisahan antara bus instruksi dan data, menambah performa ARM Cortex-M3, karena bisa membaca (*fetch*) kode dari memori program dan data dari RAM secara bersamaan (paralel). Walaupun secara manufaktur arsitektur Harvard lebih rumit, tetapi kerumitan itu tidak menjadi signifikan dibandingkan dengan performa yang diperoleh.

ARM Cortex-M3 merupakan standar *core* mikrokontroler yang lengkap, di dalamnya sudah termasuk sistem interupsi, timer Systick, sistem debug, dan pemetaan memori. Tersedia 4 Gbyte memori yang dipisahkan antara kode, SRAM (*Static RAM*), periperal dan periperal sistem. Gambar 2.1 menunjukkan core dari ARM Cortex-M3 yang dirancang oleh ARM Ltd. Perusahaan semikonduktor yang membeli lisensi dari ARM Ltd, akan menambahkan berbagai macam periperal membentuk sebuah chip mikrokontroler yang lengkap. Penambahan ini tidak ditentukan oleh ARM Ltd, tetapi tergantung kepada perusahaan yang akan memproduksi chip mikrokontroler tersebut. Penambahan ini bisa berupa ADC, DAC, UART, SPI, Ethernet (MAC), USB dan lain-lain.

Fitur akses data ARM Cortex-M3 mengijinkan adanya akses data tidak rata (*unaligned*) sehingga akan membuat pemakaian memori (SRAM) yang lebih efisien. Seperti diketahui bahwa ARM adalah prosesor 32 bit, di mana register dan SRAM memiliki lebar data 32 bit. Pada saat memproses

data ukuran 8 bit (tipe char atau byte), pada arsitektur sebelumnya satu alamat SRAM hanya akan diisi oleh 1 byte data (8 bit), dengan adanya akses tidak rata ini maka 1 byte SRAM bisa ditempati oleh 4 byte data.



Gambar 2.1 Core Prosesor ARM Cortex-M3

Sistem memori ARM Cortex-M3 juga mendukung operasi bit atau *bit-banding*. Area memori bit band ini menempati di alamat 1Mbyte dari internal SRAM dan memori periperal. Dengan adanya memori yang bisa dialami per bit ini, maka akses ke register periperal atau flag di area SRAM menjadi lebih efisien karena tidak memerlukan prosesor Boolean tambahan.

Fitur lain yang sangat penting yang dimiliki oleh prosesor ARM Cortex-M3 adalah sistem kendali interupsi yang dinamakan *Nested Vector*

Interrupt Controller (NVIC). NVIC dirancang untuk dapat menangani interupsi secara cepat, hanya dibutuhkan 12 siklus dari sinyal interupsi diterima sampai instruksi pertama di rutin layanan interupsi dieksekusi. Selain itu, sesuai dengan namanya *nested interrupt*, NVIC akan mendahulukan interupsi dengan prioritas yang lebih tinggi. Di ARM Cortex-M3, masing-masing interupsi bisa di-set dengan prioritas yang berbeda. Saat terjadi sebuah interupsi Cortex-M3 akan mengecek prioritas interupsi yang sedang ditangani saat itu, jika prioritasnya lebih rendah maka interupsi tersebut untuk sementara akan ditunda (*pre-emptive*) untuk menjalankan interupsi dengan prioritas yang lebih tinggi.

Di ARM7 dan ARM9 dikenal 2 mode instruksi yaitu instruksi ARM 32 bit dan instruksi *Thumb* 16 bit. Instruksi ARM dipakai untuk aplikasi yang membutuhkan proses dengan kecepatan tinggi sedangkan instruksi *Thumb* digunakan aplikasi yang memerlukan penghematan memori. Mode instruksi ini dipilih saat meng-compile program dan tidak bisa digunakan secara bersamaan. Di keluarga Cortex, diperkenalkan mode instruksi baru yaitu instruksi *Thumb-2* yang pada dasarnya keluarga ARM Cortex bisa menjalankan instruksi ARM 32 bit dan instruksi *Thumb* 16 bit secara bersamaan, tidak perlu memilih mode saat meng-compile program. Compiler akan otomatis memilih mana baris program yang akan di-compile dengan instruksi 32 bit dan mana yang akan di-compile dengan instruksi 16 bit. Dengan demikian instruksi *Thumb-2* akan 26% lebih hemat memori dari pada instruksi ARM dan 25% lebih cepat bila dibandingkan dengan instruksi *Thumb*.

Selain itu ARM Cortex-M3 juga dilengkapi dengan timer SysTick (*System Tick*) dan unit proteksi memori (*Memory Protection Unit - MPU*). Kedua periperal ini biasa digunakan untuk sistem aplikasi yang menggunakan sistem operasi (*Real Time Operating System - RTOS*). Timer SysTick diperlukan untuk menyediakan pewaktuan untuk *scheduler* dari RTOS sedangkan MPU akan mengatur akses memori antara aplikasi dengan OS. ARM Cortex-M3 juga menyediakan jalur untuk *debugging* program yang dinamakan *Debug Access Port* (DAP) dengan protokol JTAG (*Joint Test Action Group*) atau SWD (*Serial Wire Debug*).

2.1 MODEL PROGRAMMER

2.1.1 MODE OPERASI

ARM Cortex-M3 memiliki dua mode operasi mode

1. Mode *Thread* dan
2. Mode *Handler*

dan memiliki 2 level hak akses (*privilege*) yaitu

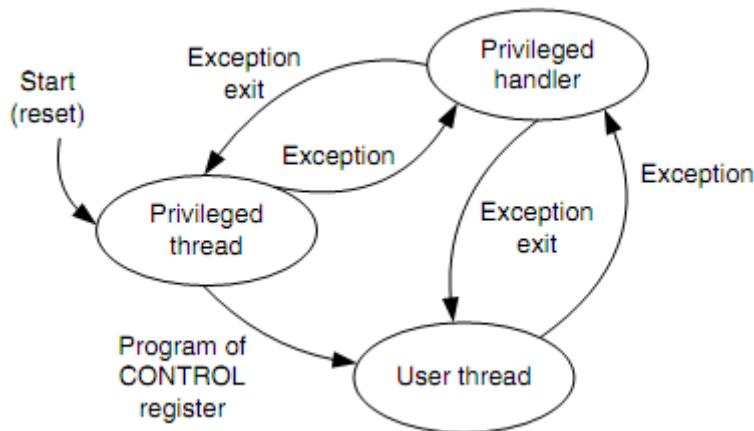
1. *unprivileged* atau *user* dan
2. *privileged*.

Mode operasi (thread atau handler) menunjukkan apakah prosesor sedang menjalankan program normal atau sedang menangani interupsi atau eksepsi (*interrupt/exception handler*). Prosesor akan berada di mode thread apabila prosesor sedang menjalankan program normal, dan berada di mode handler apabila sedang menangani sebuah interupsi atau eksepsi. Sedangkan hak akses adalah sebuah mekanisme pengaksesan memori secara aman, misal mengakses region memori kritis di OS. Saat berada di mode thread, prosesor bisa berada di level privileged atau user, sedangkan di mode handler prosesor hanya bisa di level privileged.

<i>Privileged</i>	<i>User</i>
<i>Handler mode</i>	
<i>Thread mode</i>	<i>Thread mode</i>

Gambar 2.2 Mode Operasi dan Hak Akses

Saat di level privileged, program bisa menggunakan semua instruksi dan mengakses semua area memori (kecuali area yang dikunci oleh MPU) dan bisa mengubah level dari level privileged menjadi level user melalui register CONTROL. Di level user, program tidak bisa mengubah menjadi level privileged, kecuali melalui mode handler terlebih dahulu agar register CONTROL bisa diprogram. Setelah reset prosesor akan berada di mode thread dengan level privileged. Di level user, program hanya punya akses terbatas pada instruksi MSR dan MRS, serta tidak bisa menggunakan instruksi CPS. Selain itu, akses ke timer sistem, NVIC dan blok kendali dilarang dan mungkin juga akses ke alamat memori atau periperal tertentu.



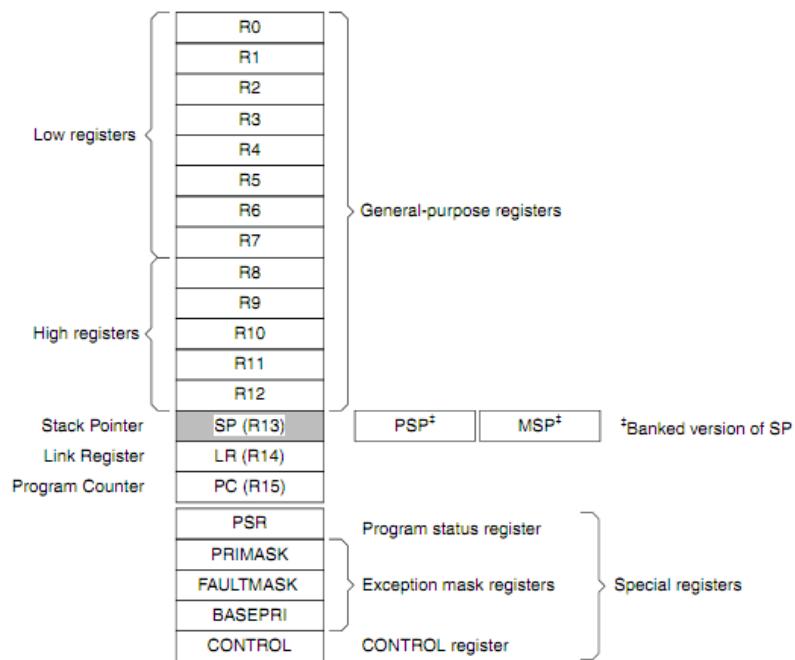
Gambar 2.3 Hak Akses hanya Bisa Diubah di Level Privileged

Keuntungan hak akses ini terlihat saat prosesor menjalankan sebuah kernel OS. Saat kernal berjalan, prosesor akan berada di level privileged, di mana semua memori bisa diakses. Ketika kemudian prosesor menjalankan sebuah aplikasi atau task, maka prosesor mengubah level menjadi user, sehingga area memori penting yang dipakai kernel, bisa dikunci agar tidak diakses oleh aplikasi tersebut.

2.1.2 REGISTER

Dalam pemrosesan data, sebuah prosesor RISC akan melakukan *load* (ambil) dan *store* (simpan). Operand (data) dari sebuah instruksi akan diambil dan disimpan di sebuah register. Register adalah tempat penyimpanan data atau memori yang berada di CPU, terpisah dari SRAM. Setelah berada di register maka data bisa diolah atau dimanipulasi dengan operasi aritmatika atau operasi lain, hasilnya kemudian diambil dari register dan disimpan kembali ke memory (SRAM). ARM Cortex-M3 dilengkapi dengan 16 register (R0 – R15) dan beberapa register khusus (gambar 2.2). Semua register memiliki lebar data 32 bits sesuai dengan arsitektur ARM.

Register yang akan diterangkan di sini adalah register yang merupakan bagian dari CPU Cortex-M3 bukan register yang digunakan untuk menakses dan mengatur berbagai macam periperal tambahan dari manufaktur semikonduktor.



Gambar 2.4 Register ARM Cortex-M3

2.1.2.1 Register Fungsi Umum

Seperti terlihat dari gambar 2.4 register fungsi umum ini terdiri dari R0 – R12. Register R0 – R7 dinamakan register rendah (*low register*) sedangkan R8 – R12 dinamakan register tinggi (*high register*). Register-register ini digunakan oleh program untuk melakukan operasi data (transfer data, operasi aritmatika atau logika).

Register rendah (R0 – R7) bisa diakses oleh semua instruksi, instruksi thumb 16 bit dan instruksi thumb-2 32 bit. Sedangkan register tinggi (R8 – R12) bisa diakses oleh semua instruksi 32 bit tetapi hanya bisa diakses oleh beberapa instruksi 16 bit. Setelah reset, isi semua register dalam kondisi acak.

2.1.2.2 Pointer Stack (R13)

Register R13 berfungsi sebagai penunjuk stack (*Stack Pointer-SP*). ARM Cortex-M3 memiliki 2 buah SP yaitu *Main Stack Pointer* (MSP) dan *Process Stack Pointer* (PSP). Kedua SP ini di-bank sehingga hanya salah satu yang aktif, diatur melalui register CONTROL. Dengan adanya 2 SP ini, sangat

berguna saat menjalankan RTOS, di mana kernel OS akan menggunakan MSP sebagai SP dan aplikasi atau task menggunakan PSP sebagai SP. Saat memasuki mode handler dengan akses privileged juga prosesor akan menggunakan MSP sebagai penunjuk memori stack.

2.1.2.3 Register Link (R14)

R14 atau Register Link (LR) digunakan untuk menyimpan alamat program (*Program Counter PC*) ketika sebuah fungsi atau sub-rutin dipanggil. Ketika sebuah fungsi dipanggil, alamat program di mana program harus kembali setelah fungsi tersebut dieksekusi, akan disimpan di register LR. Perintah assembly ARM untuk kembali dari sebuah fungsi adalah

BX LR

artinya melompat ke alamat yang ditunjuk oleh register LR.

2.1.2.4 Pencacah Program R15

R15 merupakan pencacah program (*Program Counter PC*), yang menyimpan alamat program yang sedang berjalan. Dengan memodifikasi PC, alur program bisa diatur.

2.1.2.5 Register Status Pogram (PSR)

PSR merupakan register fungsi khusus, ke 32-bit PSR dibagi 3 bagian fungsi:

1. *Application Program Status Register (APSR)*
2. *Interrupt Program Status Register (IPSR)*
3. *Execution Program Status Register (EPSR)*



Gambar 2.5 Pembagian Fungsi Register PSR

Dalam bahasa assembly ARM, register PSR bisa diakses sebagai register-individu seperti di atas atau sebagai register 32 bit penuh dengan nama PSR. Register PSR juga sering dinamakan register xPSR.

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT						Exception number

Gambar 2.6 xPSR sebagai Register 32 Bit Penuh

Register APSR berisi bendera yang menandakan status setelah sebuah instruksi dieksekusi. Tabel di bawah menjelaskan masing-masing bit dari register APSR.

Tabel 2.1 Definisi Bit Register APSR

Urutan Bit	Nama Bit	Penjelasan
31	N	Bit Negatif atau kurang dari Bernilai 0 jika operasi sebelumnya menghasilkan nilai positif, 0, lebih besar dari atau sama dengan. Bernilai 1 jika operasi sebelumnya menghasilkan nilai negatif atau kurang dari
30	Z	Bit Zero Bernilai 0 jika operasi menghasilkan nilai bukan 0 Bernilai 1 jika operasi menghasilkan nilai 0
29	C	Bit Carry (bawaan atau pinjaman) Bernilai 0 jika operasi penjumlahan tidak menghasilkan bawaan (<i>carry</i>) atau operasi pengurangan tidak memerlukan pinjaman (<i>borrow</i>) Bernilai 1 jika operasi penjumlahan menghasilkan bawaan atau operasi pengurangan memerlukan pinjaman
28	V	Bit Overflow Bernilai 0 jika operasi tidak menghasilkan overflow Bernilai 1 jika operasi menghasilkan overflow
27	Q	Bit Sticky bendera saturai Bernilai 0 jika tidak ada saturasi sejak reset atau sejak bit ini di-set ke nol Bernilai 1 jika instruksi SSAT atau USAT menghasilkan saturasi
26:0	-	Disediakan untuk pengembangan

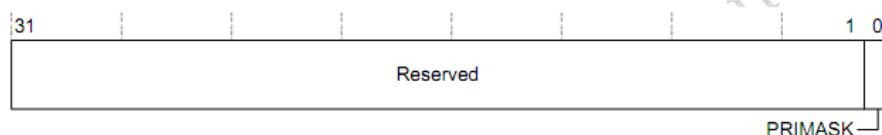
Register IPSR berisi nomer eksepsi dari rutin pelayanan interupsi (ISR) yang sedang berjalan. IPSR akan bernilai 0 jika prosesor berada dalam mode thread menjalankan program normal (tidak ada eksepsi atau interupsi). Eksespsi bisa berasal NMI (*Non-Maskable Interrupt*), manajemen memori, timer Systick atau interupsi dari periperal.

Register EPSR berisi status bit Thumb dan bit status eksekusi instruksi *If-Then* (IT) dan *Interruptible-Continuable Instruction* (ICI) atau instruksi ambil dan simpan dengan banyak operand yang terinterupsi.

2.1.2.6 Register PRIMASK, FAULTMASK dan BASEPRI

Register PRIMASK, FAULTMASK, dan BASEPRI merupakan register eksepsi yang digunakan untuk mematikan penanganan sebuah eksepsi/interupsi oleh prosesor.

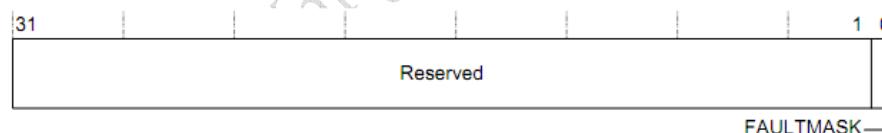
Register PRIMASK (*Priority Mask*) digunakan untuk mematikan semua interupsi atau eksepsi kecuali NMI dan eksepsi *Hard Fault*.



Gambar 2.7 Register PRIMASK

Register PRIMASK hanya bit ke-0 yang dipakai, jika bit ini di-set maka interupsi selain NMI dan hard fault akan dimatikan. Saat reset bit ini akan bernilai 0.

Register FAULTMASK digunakan untuk mematikan semua interupsi kecuali NMI. Register ini juga hanya bit ke-0 yang dipakai.



Gambar 2.8 Register FAULTMASK

Register BASEPRI (*Base Priority*) menentukan prioritas minimum untuk proses penanganan interupsi atau eksepsi. Jika register ini bernilai bukan 0, maka semua interupsi atau eksepsi dengan prioritas sama atau lebih rendah daripada nilai BASEPRI akan dimatikan.

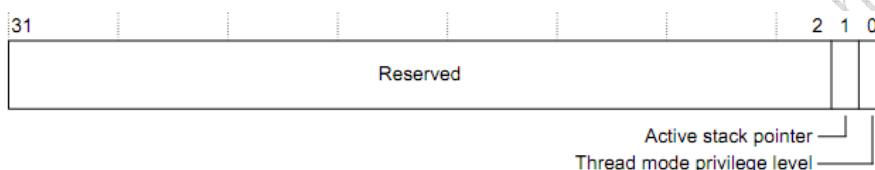


Gambar 2.9 Register BASEPRI

Register BASEPRI hanya memakai bit ke-4 sampai ke-7 (bit 7:4).

2.1.2.7 Register CONTROL

Register CONTROL digunakan untuk memilih pemakaian stack dan mode hak akses prosesor (mode privileged). Hanya 2 bit yang digunakan yaitu bit ke-0 dan bit ke-1. Bit ke-0 register ini dinamakan TPL (*Thread Mode Privileged Level*) sedangkan bit ke-1 dinamakan ASPSEL (*Active Stack Pointer Selection*).



Gambar 2.10 Register CONTROL

Bit TPL digunakan untuk memilih level privileged. TPL bernilai 0 berarti level privileged sedangkan TPL bernilai 1 berarti level non privileged (level user). Saat reset TPL akan bernilai 0. Pengubahan level privileged hanya bisa dilakukan di mode handler.

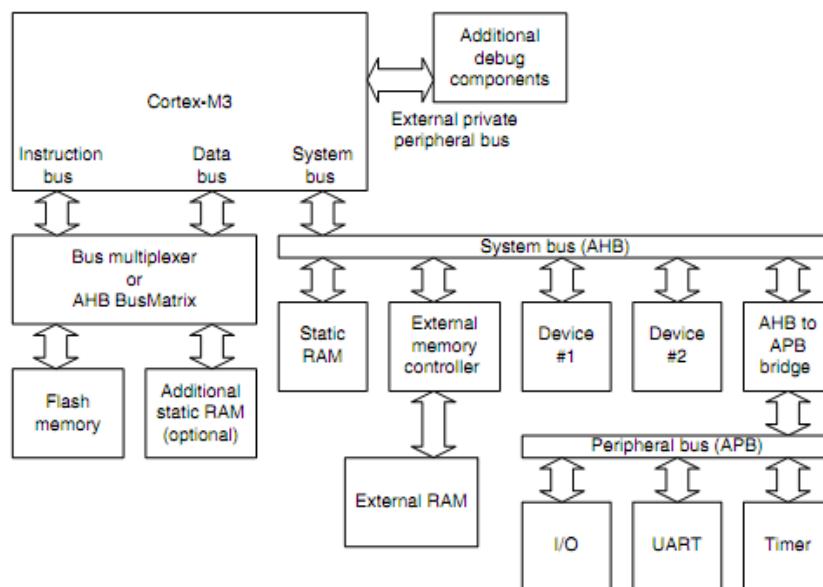
Bit ASPSEL digunakan untuk memilih SP, MSP atau PSP. ASPSEL bernilai 0 akan memilih MSP sebagai SP sedangkan ASPSEL bernilai 1 akan memilih PSP sebagai SP. Di mode handler, prosesor akan selalu menggunakan MSP, sehingga penulisan bit ASPSEL di mode handler akan diabaikan.

2.2 ARSITEKTUR BUS

Untuk menghubungkan core CPU dengan memori (flash atau SRAM), periperal internal (timer, port serial, I/O), periperal eksternal atau sistem untuk debug digunakan sistem arsitektur bus yang dinamakan dengan AMBA (*Advanced Microcontroller Bus Architectur*). AMBA merupakan standar yang dikembangkan ARM Ltd. untuk interkoneksi dalam chip dan manajemen fungsi di dalam sebuah mikrokontroler atau *System-On-Chip*.

Chip (SoC). ARM Cortex-M3 menggunakan dua jenis sistem bus yang sesuai dengan spesifikasi AMBA yaitu AHB-Lite dan APB.

AHB (*Advanced High-performance Bus*) Lite digunakan untuk interkoneksi dengan kecepatan tinggi, misalnya interkoneksi dengan memori flash atau SRAM. Sedangkan APB (*Advanced Peripheral Bus*) digunakan untuk interkoneksi dengan kecepatan rendah misalnya register untuk mengendalikan periperal internal, timer misalnya. Gambar 2.11 menunjukkan interkoneksi di dalam prosesor Cortex-M3.



Gambar 2.11 Blok Diagram ARM Cortex-M3

ARM Cortex-M3 memiliki 4 buah bus yang menghubungkan core CPU dengan memori, periperal dan perangkat luar, yaitu

1. Bus instruksi (*I-Code Bus*) digunakan oleh core untuk mengambil (*fetch*) instruksi dari memori flash atau SRAM dalam rentang alamat memori 0x00000000 sampai 0x1FFFFFFF. Bus ini menggunakan bus AHB.
2. Bus data (*D-Code Bus*) digunakan untuk mengakses data dan juga instruksi dari memori flash atau SRAM dengan rentang memori 0x00000000 sampai 0x1FFFFFFF. Bus ini menggunakan bus AHB.

3. Bus sistem (*S Bus*), digunakan mengakses data dari SRAM atau periperal dalam rentang memori 0x20000000 sampai 0xDFFFFFFF dan 0xE0100000 sampai 0xFFFFFFFF. Untuk periperal dengan kecepatan tinggi digunakan bus AHB, sedangkan untuk periperal dengan kecepatan rendah digunakan APB setelah melalui jembatan AHB ke APB. Akses instruksi juga bisa dilakukan melalui bus ini, tetapi tidak seefisien melalui I-Code.
4. Bus eksternal, menggunakan bus APB, untuk mengakses periperal eksternal dan debug.

Arsitektur Harvard yang digunakan memungkinkan akses data dan instruksi secara bersamaan, tetapi jika akses data dan instruksi melalui multiplexer, di SRAM tambahan, maka akses tidak dapat dilakukan secara bersamaan. Selain sebagai memori data, SRAM juga bisa digunakan untuk menyimpan program yang akan dijalankan. Misalnya saat pengembangan program, untuk mengurangi siklus tulis dan hapus memori flash. Tetapi efisiensinya lebih rendah bila dibandingkan dengan menjalankan program dari memori flash.

Perangkat-perangkat yang membutuhkan akses dengan kecepatan tinggi, seperti SRAM atau kendali memori eksternal, akan langsung terhubung dengan bus sistem (AHB). Sedangkan perangkat dengan kebutuhan yang lebih rendah akan terhubung ke APB yang kemudian melalui sebuah pengantara (*bridge*) akan menghubungkannya ke AHB.

2.3 PIPELINE

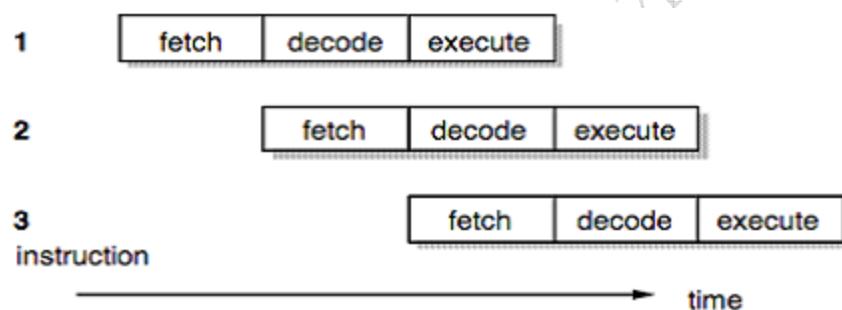
Mikrokontroler dalam mengeksekusi program dari memori program akan melakukan setidaknya 3 tahap berikut:

1. *Fetch*, pengambilan/pembacaan instruksi dari memori program
2. *Decoding*, pengjemahan instruksi
3. *Execution*, eksekusi instruksi.

Pada beberapa mikrokontroler, misalnya keluarga 8051, tahapan tersebut dilakukan secara berurutan (sekuensial), sebuah instruksi harus dieksekusi sampai selesai sebelum mengambil instruksi selanjutnya, sehingga sebuah instruksi memerlukan beberapa siklus untuk dieksekusi.

Untuk meningkatkan jumlah instruksi yang bisa dieksekusi per siklus waktu atau untuk mengurangi waktu untuk mengeksekusi sebuah instruksi, arsitektur sebuah prosesor menggunakan sebuah teknik yang dinamakan dengan teknik *instruction pipeline*. Teknik pipeline ini akan membuat tahap-tahap instruksi di atas bisa dilakukan secara bersamaan (paralel) dengan menggunakan multi pipeline. Misal mikrokontroler AVR dan PIC menggunakan pipeline 2 tingkat, sedangkan Intel Pentium 4 menggunakan pipeline 20 tingkat. Sedangkan mikrokontroler ARM Cortex-M3 menggunakan 3 pipeline.

Secara sederhana pipeline 3 tingkat bisa dijelaskan sebagai berikut: siklus pertama, pipeline tingkat pertama melakukan pengambilan instruksi (*fetching*), pipeline tingkat kedua dan ketiga dalam keadaan siap. Siklus selanjutnya, pipeline pertama akan melakukan penerjemahan instruksi (*decoding*), sedangkan pipeline kedua melakukan pengambilan instruksi. Siklus berikutnya, pipeline pertama mengeksekusi instruksi dan pipeline kedua menerjemahkan instruksi sedangkan pipeline ketiga mulai mengambil instruksi dari memori program.



Gambar 2.12 Pipeline 3 Tingkat

Idealnya antar tingkat pipeline harus seimbang, artinya setiap tingkat harus melakukan eksekusi dalam waktu yang sama, jika tidak keseluruhan proses akan terpengaruh.

Saat mengeksekusi instruksi percabangan (*branch*), pipeline harus dikosongkan terlebih dahulu untuk kemudian diisi lagi saat pengambilan instruksi di alamat tujuan percabangan. ARM Cortex dilengkapi dengan fasilitas prediksi percabangan (*branch prediction*), sehingga saat mengeksekusi instruksi percabangan, prosesor akan melakukan

pengambilan instruksi secara spekulatif. Tapi jika pengambilan spekulatif tidak bisa dilakukan, seperti saat instruksi percabangan tidak langsung (*indirect branch*), pipeline akan dikosongkan terlebih dahulu.

2.4 MODEL MEMORI

ARM Cortex-M3 menggunakan pengalamanan 32 bit, sehingga bisa mengalami sampai dengan 4 Gbyte (4294967296 byte). Walaupun menggunakan arsitektur Harvard, di mana ada pemisahan antara program dan data, bukan berarti Cortex-M3 memiliki 2x4 GB alamat, seperti pada mikrokontroler 8051 yang memiliki 64 KB alamat untuk masing-masing program dan data. Di ARM Cortex-M3, program dan data keduanya berada di dalam alamat 4 GB tersebut, pemisahannya dilakukan oleh sistem bus internal Cortex-M3.

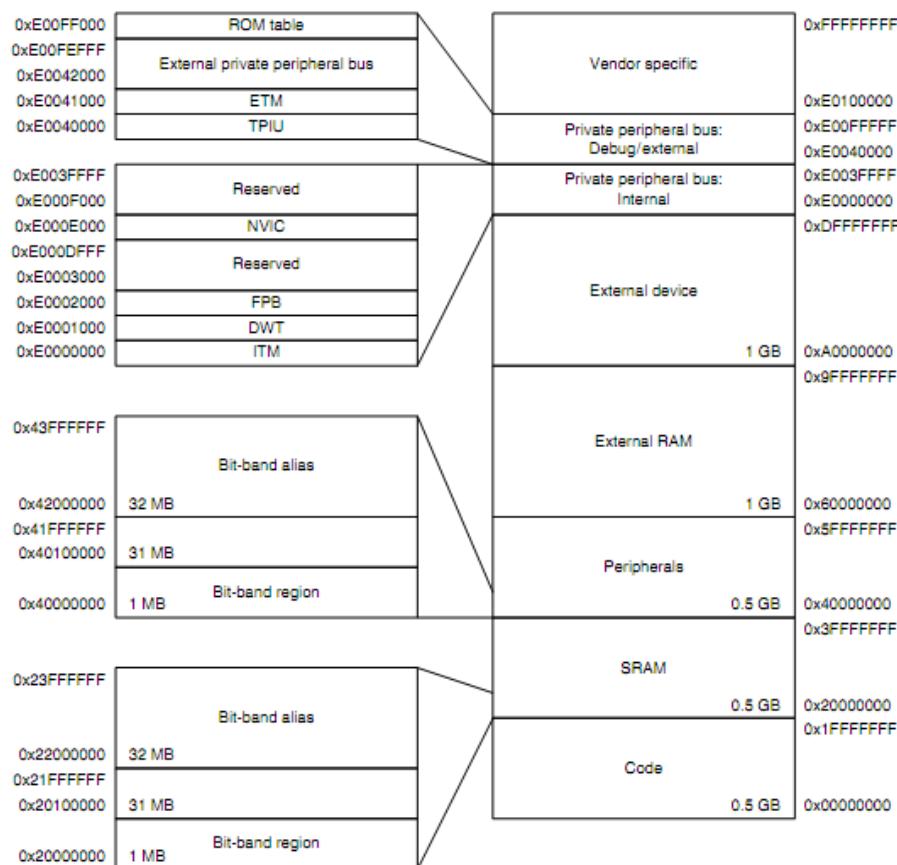
2.4.1 PEMETAAN MEMORI

Ruang memori 4 GB dipetakan dalam berapa area memori dengan alamat yang telah ditentukan untuk masing-masing area. Pemetaan memori ARM Cortex-M3 ditunjukkan oleh gambar 2.13.

Alamat 1 GB pertama ditempati oleh area untuk kode dan SRAM. Ruang untuk kode akan terhubung dengan bus I-Code dan SRAM tentunya akan terhubung dengan bus D-Code. Walapun sebenarnya kode juga bisa disimpan dan dieksekusi dari SRAM, namun waktu eksekusi menjadi lebih lambat. Alamat 1 MB pertama dari SRAM bisa dialamat dengan operasi *bit-band* untuk melakukan operasi manipulasi bit.

Area berikutnya (0.5 GB) adalah area untuk periperal internal yang akan disematkan ke mikrokontroler oleh perusahaan semikonduktor yang telah membeli lisensi dari ARM. Di area alamat ini, register-register untuk mengatur fungsi dari periperal internal berada. Periperal yang akan dipasang akan berbeda-beda, tapi setidaknya akan ada GPIO (*General Purpose Input/Output*), USART, Timer, ADC, DAC, RTC (*Real Time Clock*) dan lain-lain. Mikrokontroler dengan spesifikasi lebih tinggi mungkin akan ada penambahan USB atau ethernet. Seperti halnya SRAM, alamat awal sebesar 1 MB juga merupakan area bit band. Sehingga memudahkan untuk mengendalikan, misalnya, GPIO per bit tanpa perlu

melakukan operasi baca-salin-tulis. Operasi bit band akan dijelaskan di sub bab terpisah.



Gambar 2.13 Pemetaan Memori ARM Cortex-M3

Area memori berikutnya diperuntukan untuk memori (RAM) atau peralatan eksternal, masing-masing 1 GB. Area RAM eksternal bisa juga untuk menyimpan dan mengeksekusi program, sedangkan area untuk peralatan eksternal tidak bisa.

Area terakhir (0.5 GB) digunakan sebagai alamat untuk *Private Peripheral Bus* (PPB) internal dan debug/eksternal. PPB internal akan menangani sistem interupsi (NVIC), unit FPB (*Fetch Patch and Breakpoint*), DWT (*Data Watchpoint and Trace*) dan ITM (*Instrumentation Trace Macrocell*). Sedangkan PPD debug/eksternal akan menangani unit ETM (*Embedded*

Trace Macrocell) dan unit TPIU (*Trace Port Interface Unit*). Di area ini juga para manufaktur bisa menambahkan peralatan yang khusus.

2.4.2 AKSES MEMORI

Peta memori yang ditunjukan oleh gambar 2.13 di atas memberikan gambaran tentang apa saja yang berada di area masing-masing. Selain itu juga menunjukan sifat (*attribute*) bagaimana mengakses area memori tersebut. ARM Cortex-M3 memiliki beberapa atribut ketika mengakses memori:

1. *Bufferable*, menulis ke memori dilakukan dengan menulis ke buffer, sementara prosesor bisa mengeksekusi instruksi selanjutnya
2. *Cacheable*, data yang didapat dari pembacaan memori dapat disimpan ke sebuah *cache* memori, sehingga pada saat akses selanjutnya data bisa dibaca dari *cache* memori tersebut sehingga bisa meningkatkan kecepatan dalam eksekusi program.
3. *Executable*, prosesor bisa mengambil (*fetch*) dan mengeksekusi program dari area memori ini.
4. *Sharable*, data dari area memori ini bisa dipakai bersama oleh beberapa master bus.

Cortex-M3 akan memberikan informasi tentang atribut memori sistem memori untuk setiap instruksi atau saat transfer data. Atribut memori bisa diganti dengan menggunakan MPU, jika tersedia. Walaupun Cortex-M3 tidak memiliki memori cache atau kendali cache, tetapi manufaktur mikrokontroler bisa menambahkan sebuah unit cache untuk menentukan atribut akses memori.

Tabel 2.2 menunjukan atribut default untuk masing-masing area memori. Jika MPU tersedia, maka atribut tersebut bisa dikonfigurasi melalui MPU tersebut. Ada tidaknya MPU tergantung dari manufaktur mikrokontroler masing-masing.

Selain atribut memori, hak untuk mengakses area memori melalui program user (mode user) juga telah ditentukan, apakah sebuah area memori bisa diakses atau tidak (diblok). Dan hak akses ini juga bisa diatur melalui MPU jika tersedia. Pengaturan ini untuk mencegah program yang berjalan di mode user (*non privileged*) dari mengakses

sistem memori seperti NVIC atau memori yang dipakai oleh sebuah sistem operasi (RTOS).

Tabel 2.2 Atribut Memori

Area Memori	Alamat	Fungsi	Cacheable	Executable, Bufferable
Code	0x00000000-0x1FFFFFFF	Program dan data	Cacheable	Executable, bufferable
SRAM	0x20000000-0x3FFFFFFF	RAM internal	cacheable	Executable, bufferable
Periperal	0x40000000-0x5FFFFFFF	Periperal internal	tidak cacheable	Tidak executable
Eksternal RAM	0x60000000-0x7FFFFFFF	Memori eksternal	cacheable	Executable
Eksternal RAM	0x80000000-0x9FFFFFFF	Memori eksternal	cacheable	Executable
Periperal Eksternal	0xA0000000-0xBFFFFFFF	Eksternal periperal	Tidak cacheable	Tidak executable, tidak bufferable
Periperal Eksternal	0xE0000000-0xFFFFFFFF	PPB dan vendor-spesific	Tidak cacheable	Tidak executable, tidak bufferable (vendor-specific bufferable)

Tabel 2.3 Hak Akses Memori

Area Memori	Alamat	Hak Akses
Vendor specific	0xE0100000-0xFFFFFFFF	Akses penuh
ETM	0xE0041000-0xE0041FFF	Diblok
TPIU	0xE0040000-0xE0040FFF	Diblok

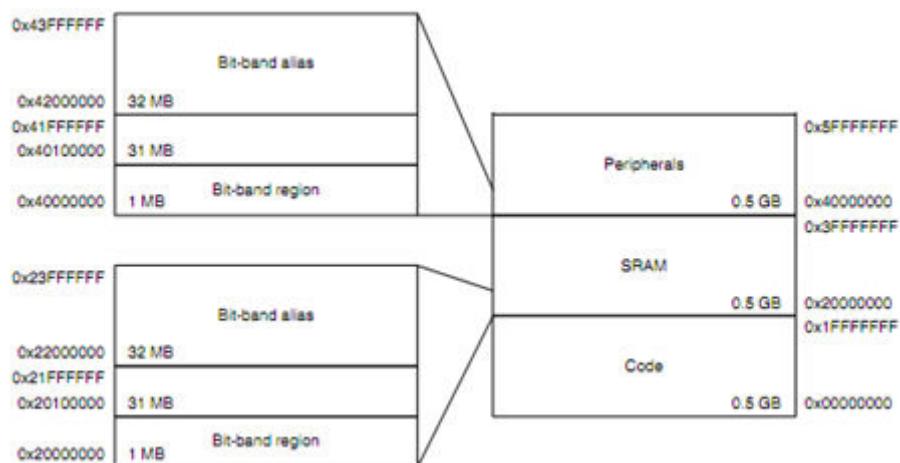
PPB Internal	0xE000F000–0xE003FFFF	Diblok
NVIC	0xE000E000–0xE000EFFF	Diblok
FPB	0xE0002000–0xE0003FFF	Diblok
DWT	0xE0001000–0xE0001FFF	Diblok
ITM	0xE0000000–0xE0000FFF	Hanya baca
RAM Eksternal	0x60000000–0x9FFFFFFF	Akses penuh
Periperal	0x40000000–0x5FFFFFFF	Akses penuh
SRAM	0x20000000–0x3FFFFFFF	Akses penuh
Code	0x00000000–0x1FFFFFFF	Akses penuh

2.4.3 OPERASI BIT BAND

ARM Cortex-M3 tidak menyediakan instruksi khusus operasi bit untuk memori (RAM) atau pun periperal. Instruksi bit khusus akan menambah ukuran dan kerumitan CPU, hal yang dihindari oleh arsitektur ARM Cortex. ARM mengatasi hal ini dengan sebuah teknik yang dinamakan dengan *bit banding* yang mengijinkan manipulasi bit secara langsung di area memori (SRAM) dan periperal tanpa harus menggunakan instruksi bit khusus. Prinsipnya hampir sama dengan mikrokontroler keluarga 8051 yang mempunyai RAM internal (alamat 0x20 – 0x2F) yang bisa diakses per bit, bedanya 8051 mempunyai instruksi khusus untuk operasi bit untuk mengakses RAM tersebut dan periperal yang dimilikinya.

Seperti telah dijelaskan di sub-bab sebelumnya, ARM Cortex-M3 menyediakan area memori untuk operasi bit band yaitu di area SRAM dan periperal, masing-masing 1 MB dari alamat awal sebagai area memori untuk operasi bit dan 32 MB sebagai alamat alias. Area 32 MB ini dipakai untuk proses *remapping* dari alamat 1 MB tersebut, artinya alamat bit dari area 1MB akan dipetakan (*remapping*) ke alamat 32 MB. Dengan operasi bit band ini, bit ke-0 (LSB) dari alamat RAM 0x20000000 akan

dipetakan ke alamat alias 0x22000000, bit ke-1 akan dipetakan ke alamat 0x22000004, bit ke-2 akan dipetakan ke alamat 0x22000008 dan seterusnya.



Gambar 2.14 Memori Bit-Band

Rumus berikut bisa digunakan untuk memperoleh alamat alias dari bit yang dimaksud:

$$\text{bit_word_addr} = \text{bit_band_base} + \text{bit_word_offset}$$

$$\text{bit_word_offset} = (\text{byte_offset} \times 32) + (\text{bit_number} \times 4)$$

di mana:

`bit_word_offset` adalah posisi dari bit target di area memori bit band.

`bit_word_addr` adalah alamat dari area alias yang memetakan ke bit target (alamat alias yang dicari).

`bit_band_base` adalah alamat awal dari area memori alias, yaitu 0x22000000 untuk SRAM dan 0x42000000 untuk periperal.

`byte_offset` adalah offset dari alamat bit band dari bit target yang dicari terhadap alamat awal area bit band.

`bit_number` adalah posisi, 0 – 31, dari bit target.

Sebagai contoh rumus di atas dipakai untuk menentukan alamat alias dari bit ke-0 alamat 0x20000000. Maka

$$\text{byte_offset} = 0x20000000 - 0x20000000 = 0x0$$

bit_word_offset = (0x32) + (2x4) = 8

bit_word_addr = 0x22000000 + 8 = 0x22000008

Dengan adanya operasi bit band ini, untuk melakukan operasi bit, misalnya mengendalikan sebuah LED yang terhubung ke bit ke-0 dari sebuah port, tidak perlu dengan melakukan operasi baca-ubah-tulis (*read-modify-write*). Artinya ketika akan menghidupkan atau mematikan LED tersebut, pertama port tersebut dibaca dulu, kemudian dengan operasi logika AND atau OR, bit yang bersangkutan diubah, kemudian hasilnya dikirimkan kembali ke alamat port tersebut. Hal ini membutuhkan siklus instruksi yang lebih panjang.

Contoh program dalam bahasa assembly berikut akan mengubah bit ke-2 dari alamat RAM 0x20000000 dengan metode baca-ubah-tulis dan metode bit band. Bit ke-0 tersebut akan dipetakan ke alamat alias 0x22000008.

Program dengan metode baca-ubah-tulis:

```
LDR      R0, = 0x20000000 ;Alamat memori
LDR      R1, [R0]          ;baca
ORR.W   R1, #0x4          ;ubah
STR      R1, [R0]          ;tulis kembali ke RAM
```

Program dengan operasi bit band

```
LDR      R0, = 0x22000008 ;Alamat bit telah dipetakan
MOV      R1, #1            ;bit di-set
STR      R1, [R0]          ;tulis bit
```

Dan program berikut merupakan contoh program dalam bahasa assembly untuk membaca bit ke-2 dari alamat 0x20000000.

Program tanpa metode bit band

```
LDR      R0, = 0x20000000 ;Alamat memori
LDR      R1, [R0]          ;baca
UBFW.W  R1, #2, #1        ;baca bit ke-2
```

Program dengan bit band

```
LDR      R0, = 0x22000008 ;Alamat yang telah dipetakan
LDR      R1, [R0]          ;baca bit ke-2
```

Dari kedua contoh program di atas terlihat bahwa, dengan menggunakan operasi bit band, program akan lebih pendek sehingga eksekusi juga pasti akan lebih cepat.

Bagaimana jika menggunakan bahasa C? Pada dasarnya tidak secara langsung mendukung operasi bit band ini. Karena compiler C tidak tahu bahwa sebuah alamat memori bisa diakses melalui sebuah alamat memori alias. Cara yang paling mudah adalah dengan mendefinisikan secara terpisah antara alamat memori dengan alamat aliasnya. Misal, sebuah PORTB (PB) beralamat di 0x40000000, sehingga PB0 (bit ke-0) akan berada di alamat 0x42000000 dan PB1 akan memiliki alamat alias di 0x42000004. Definisinya dalam bahasa C adalah sebagai berikut:

```
#define PB    *((volatile unsigned long *)) (0x40000000)\n#define PB0   *((volatile unsigned long *)) (0x42000000)\n#define PB1   *((volatile unsigned long *)) (0x42000004))
```

Untuk mengakses PB, dengan alamat normal bisa dilakukan dengan

```
PB = 0xAB;
```

dan untuk men-set PB1 tanpa menggunakan bit band dilakukan dengan operasi OR

```
PB = PB | 0x2;
```

sedangkan untuk mengakses PB1 dengan operasi bit band, bahasa C-nya akan menjadi seperti di bawah ini

```
PB1 = 0x1;
```

Bisa juga dengan membuat sebuah makro dalam bahasa C, agar mempermudah untuk mendapatkan alamat alias dari bit yang bersangkutan, dengan menggunakan rumus yang telah dijelaskan sebelumnya

```
//Makro untuk mengubah alamat bit band dan nomor bit ke\nalamat alias\n#define BITBAND(addr, bitnum) ((addr & 0xF0000000) +\n                           0x20000000 + (addr & 0xFFFF) <<5)+(bitnum <<2))\n//Ubah alamat sebagai pointer\n#define MEM_ADDR(addr) *((volatile unsigned long *)(addr))
```

Dengan menggunakan definisi makro ini, contoh program untuk mengakses PB bisa diubah sebagai berikut:

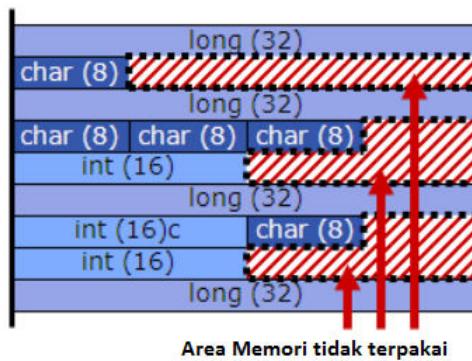
```
#define PB 0x40000000\n#define BITBAND(addr, bitnum) ((addr & 0xF0000000) +\n                           0x20000000 + (addr & 0xFFFF) <<5)+(bitnum <<2))\n#define MEM_ADDR(addr) *((volatile unsigned long *)(addr))\n\n//Akses memori secara normal\nMEM_ADDR(PB) = 0xAB;
```

```
//Men-set bit ke-1 tanpa bit band
MEM_ADDR(PB) = MEM_ADDR(PB) | 0x2;
//Set bit ke-1 dengan bit band
MEM_ADDR(BITBAND(PB,1)) = 0x1;
```

Saat membuat makro, harus dipastikan bahwa jenis data yang digunakan adalah volatile, sehingga setiap kali variabel tersebut diakses, data benar-benar diambil dari alamat memori yang bersangkutan.

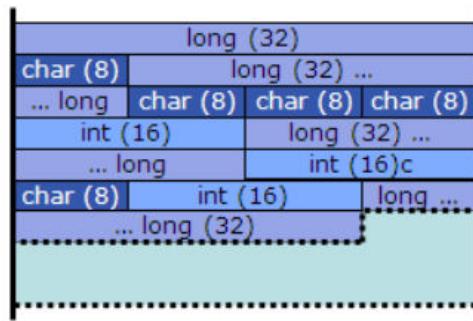
2.4.4 AKSES MEMORI TAK RATA

ARM adalah prosesor 32 bit, dengan lebar bus data 32 bit, sehingga memori (RAM) yang digunakan juga mempunyai lebar data 32 bit untuk performa maksimal. Dalam aplikasi sebenarnya, tipe data bisa menggunakan byte (8 bit), interger (16 bit) atau long (32 bit). Dalam terminologi ARM, 32 bit dinamakan *word* dan 16 bit dinamakan *half word*. Prosesor ARM sebelumnya yang hanya mengenal akses memori rata (*aligned access*) saat menangani tipe data byte (8 bit) akan mengalami pemborosan memori, karena lebar memori 32 bit hanya terpakai 8 bit.



Gambar 2.15 Akses Memori Rata

ARM Cortex mengatasi hal ini dengan menggunakan akses memori tak rata (*unaligned access*). Dengan mode akses tak rata ini, 1 alamat 32 bit bisa menyimpan 4 variabel data 8 bit, sehingga memori (SRAM) bisa digunakan secara efisien.



Gambar 2.16 Akses Memori Tak Rata

Akses memori tak rata ini bisa dilakukan pada akses memori normal, seperti instruksi LDR, LDRH, STR dan STRH. Tetapi ada beberapa pengecualian:

1. Transfer tak rata tidak bisa dilakukan di instruksi *Store/Load* ganda.
2. Operasi stack (PUSH/POP) harus menggunakan akses rata.
3. Akses eksklusif (misalnya LDREX atau STREX) harus dilakukan menggunakan akses rata, jika tidak akan terjadi eksepsi (*usage fault*).
4. Operasi bit band juga tidak bisa dilakukan secara tak rata.

Walaupun akses memori tak rata akan membuat pemakaian SRAM menjadi lebih efisien, namun saat melakukan operasi transfer tak rata, akan membagi transfer ini menjadi beberapa bagian, sehingga akan menggunakan siklus clock lebih banyak dan akan menjadi lebih lambat.

2.5 MODEL EKSEPSI

Dalam pemrograman eksepsi (*exception*) bisa diartikan sebagai kejadian (*event*) yang mengganggu alur normal sebuah program, yang diakibatkan terjadinya kesalahan saat program dijalankan (*run time error*). Misalnya sebuah program pembagian 2 buah bilangan bulat, ketika pembaginya bernilai nol , maka akan terjadi eksepsi *divide by zero*. Pada saat ini, jika tidak ada fungsi untuk menanganinya (*exception handler*), maka sistem operasi komputerlah yang akan menanganinya, misalnya dengan menampilkan pesan kesalahan pada monitor.

Pada mikrokontroler, program yang harus dijalankan dibaca dari memori program (memori flash). Alamat memori program yang harus dibaca disimpan di register PC. Selama alur normal, PC akan dinaikkan secara beraturan. Kecuali ketika program harus memanggil atau melompat ke

sebuah fungsi atau sub-rutin, maka isi dari PC akan diubah menjadi alamat dari fungsi tersebut. Perubahan ini secara “sadar” dilakukan oleh program, artinya di program tertera jelas kapan dan di mana proses ini dilakukan. Ada juga proses pengubahan alur program yang bisa terjadi secara tiba-tiba dan dipicu oleh hardware atau periperal internal. Inilah yang dinamakan interupsi. Pada saat terjadi interupsi, program akan melompat ke sebuah sub-rutin yang dinamakan dengan ISR (*Interrupt Service Routine*). Walaupun demikian, sebuah interupsi tetap harus diinisialisasi di awal program, periperal mana saja yang akan menginterupsi CPU. Interupsi adalah salah satu jenis eksepsi.

ARM Cortex-M3 mengenal 3 jenis eksepsi:

1. Eksepsi karena interupsi yang dibangkitkan oleh periperal internal, dinamakan dengan IRQ (*Interrupt Request*). Penanganannya dilakukan oleh ISR.
2. Eksepsi yang dibangkitkan oleh kesalahan (*fault*). Eksepsi ini dibangkitkan oleh kesalahan saat akses memori, kesalahan di sistem bus, atau kesalahan instruksi. Penanganannya dilakukan oleh *fault handler*.
3. Eksepsi sistem, yang dibangkitkan oleh Non-Maskable Interrupt (NMI) atau software yang berhubungan dengan sistem operasi. Fungsi penanganannya dinamakan *system handler*.

ARM Cortex-M3 mempunyai eksepsi kesalahan dan eksepsi sistem dari nomor 1-15 sedangkan interupsi eksternal dimulai dari nomor 16. Level prioritas dari eksepsi ada yang bisa diprogram ada juga yang tidak, sedangkan semua interupsi eksternal prioritasnya bisa diprogram. ARM Cortex-M3 bisa dilengkapi dengan interupsi eksternal dari 1-240, tergantung kepada pembuat SoC dan untuk keperluan apa prosesor diproduksi. Reset merupakan eksepsi yang terjadi saat pertama kali daya dinyalakan (power up) atau dengan memberi sinyal ke pin reset ketika program sedang berjalan (warm reset). Sedangkan NMI bisa dipicu oleh periperal atau software, dan secara permanen selalu aktif dengan prioritas yang tetap.

Tabel 2.4 Eksepsi dan Interupsi ARM Cortex-M3

No. Eksepsi	Tipe Eksepsi	Prioritas	Keterangan
1	Reset	-3	Reset

		(Tertinggi)	
2	NMI	-2	Non Maskable Interrupt
3	Hard fault	-1	Semua kondisi kesalahan (fault) jika handler dari fault yang lain tidak diaktifkan
4	MemManage Fault	Dapat diprogram	Kesalahan di manajemen memori, disebabkan oleh kesalahan di MPU atau akses ilegal, misalnya pengambilan instruksi dari area memori yang tidak bisa mengeksekusi program
5	Bus Fault	Dapat diprogram	Error di bus sistem, terjadi ketika antarmuka AHB menerima tanggapan error dari sebuah bus slave (disebut juga <i>prefetch abort</i> jika merupakan pengambilan instruksi atau <i>data abort</i> jika error di akses data)
6	Usage Fault	Dapat diprogram	Error yang disebabkan karena instruksi yang tidak terdefinisi, instruksi coprosesor (Cortex-M3 tidak mempunyai coprosesor, masalah saat kembali dari interupsi dan akses tak rata menggunakan instruksi multi simpan dan ambil)
7 - 10	Tidak dipakai		
11	SVC	Dapat diprogram	Supervisor Call, berhubungan dengan sistem operasi (RTOS)
12	Debug monitor	Dapat diprogram	Monitor debug (<i>break point</i> , <i>watchpoint</i> , atau permintaan debug dari luar)
13	Tidak dipakai		
14	PendSV	Dapat diprogram	Pendable Service Call, berhubungan dengan RTOS
15	SYSTICK	Dapat diprogram	Timer System Tick
16 - 255	Interupsi eksternal	Dapat diprogram	Eksepsi karena interupsi dari periperal internal, tergantung dari sistem chip yang dibuat

2.5.1 HARD FAULT

Hard fault merupakan eksepsi yang terjadi ketika prosesor tidak bisa mengeksekusi fungsi handler dari eksepsi yang lain (*MemManage fault*, *Usage fault*, dan *Bus fault*), bisa karena fungsi handler tidak didefinisikan

atau eksepsinya tidak diaktifkan. HardFault mempunyai prioritas tertinggi setelah reset dan NMI.

Oleh karena Hard fault bisa dipicu oleh eksepsi yang lain, maka untuk mengetahui eksepsi mana yang sebenarnya memicu Hardfault bisa diketahui dari register status Hard fault (HFSR = *Hard Fault Status Register*) yang beralamat di 0xE000ED2C. Seperti yang ditunjukan oleh tabel di bawah.

Tabel 2.5 Register HFSR (0xE000ED2C)

Bit	Nama	Sifat	Nilai Reset	Keterangan
31	DEBUGEVT	Baca/Tulis	0	Hard fault disebabkan oleh debug
30	FORCED	Baca/Tulis	0	Hard fault dipicu oleh bus fault, MemManage fault, atau Usage fault
29:2	-	-	-	-
1	VECTBL	Baca/Tulis	0	Hard fault disebabkan oleh gagalnya fetch
0	-	-	-	-

2.5.2 MEMMANAGE FAULT

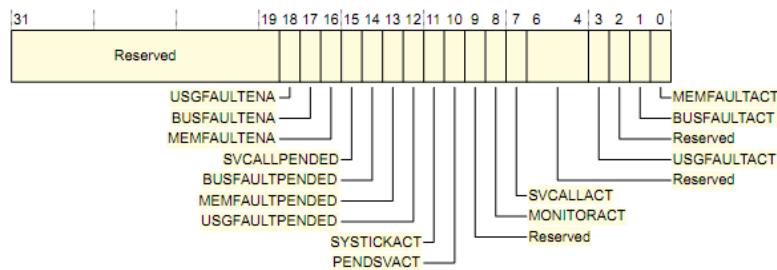
MemManage (Memory Management) fault, bisa dipicu karena adanya kesalahan yang berhubungan dengan MPU atau adanya akses ilegal terhadap memori (misalnya mengeksekusi kode dari area memori yang tidak bisa dieksekusi). Kemungkinan-kemungkinan yang bisa memicu eksepsi MemManage yang berhubungan dengan MPU, misalnya:

1. Akses ke area memori yang tidak didefinisikan oleh MPU
2. Menulis ke area yang hanya bisa dibaca
3. Akses dari mode user ke area memori yang hanya bisa diakses saat mode privileged

Ketika terjadi MemManage Fault, dan fungsi penanganannya diaktifkan, maka prosesor akan memanggil fungsi tersebut, sedangkan jika tidak prosesor akan membangkitkan eksepsi Hard Fault dan akan men-set bit FORCED (bit ke-30) register HFSR. Level prioritas MemManage fault ini bisa di set. Level prioritas ini berguna karena ARM Cortex-M3, seperti yang akan dijelaskan, mengenal interupsi/eksepsi bersarang (*nested*).

Ketika terjadi sebuah interupsi/eksepsi, maka prosesor Cortex-M3 akan mengecek dan membandingkan level interupsi ini, ketika pada saat terjadinya eksepsi ada eksepsi lain yang sedang dikerjakan dengan level interupsi yang sama atau lebih tinggi, maka eksepsi yang baru terjadi tersebut akan ditunda terlebih dahulu. Tetapi jika level prioritas eksepsi yang sedang terjadi lebih rendah, maka prosesor akan menghentikan dulu proses eksepsi tersebut dan mengerjakan eksepsi yang baru terjadi.

Eksepsi MemManage fault diaktifkan melalui bit MEMFAULTENA (bit ke-16) di register SHCSR (*System Handler Control and State Register*). Register SHCSR juga mengendalikan pengaktifan dari eksepsi-eksepsi yang lain, kecuali Hard fault. SHCSR beralamat di 0xE000ED24.



Gambar 2.17 Register SHCSR

Bit-bit dengan nama berakhiran ENA, merupakan bit untuk mengaktifkan atau mematikan eksepsi (ENABLE), jika bit di-set (bernilai 1) maka eksepsi aktif, jika di-clear (bernilai 0) eksepsi tidak aktif. Bit-bit yang berakhiran PENDED merupakan bit-bit yang menunjukkan bahwa eksepsi tersebut sedang ditunda (*pending*). Sedangkan bit-bit yang berakhiran ACT, menunjukkan bahwa eksepsi sedang dalam kondisi aktif jika bernilai 1 (di-set).

Kondisi atau penyebab dari eksepsi MemManage ditunjukan oleh register MMFSR (*Memory Management Fault Status Register*). MMFSR merupakan bagian dari register CFSR (*Configurable Fault Status Register*). Isi dari register MMFSR ditunjukan oleh tabel di bawah.



Gambar 2.18 Register CFSR

Ketika terjadi eksepsi MemManage fault, penyebab utamanya bisa dilihat di bit MSTKERR, MUSTKERR, DACCVILO atau IACCVIOL yang menunjukkan apakah eksepsi disebabkan saat proses di memori stack (*stacking* atau *unstacking*), saat akses ke memori data atau kesalahan saat akses ke instruksi. Jika bit MMARVALID di-set, maka lokasi memori yang menyebabkan eksepsi ini bisa dilihat di register MMAR (*Memory Management Address Register*).

Tabel 2.6 Register MMFSR

Bit	Nama	Sifat	Nilai Reset	Keterangan
7	MMARVALID	-	0	Jika di-set menunjukkan bahwa register MMAR akan berisi alamat memori yang menyebabkan eksepsi MemManage.
6:5	-	-	-	-
4	MSTKERR	Baca/Tulis	0	Error saat penyimpanan di stack.
3	MUSTKERR	Baca/Tulis	0	Error saat pengambilan dari stack
2	-	-	-	-
1	DACCVIOL	Baca/Tulis	0	Pelanggaran saat akses ke memori data
0	IACCVIOL	Baca/Tulis	0	Pelanggaran saat akses instruksi

2.5.3 BUS FAULT

Bus fault bisa terjadi ketika ada kesalahan atau error saat terjadi transfer di antarmuka AHB, baik saat pengambilan instruksi, yang disebut *prefetch abort* atau saat baca/tulis di memori data yang disebut *data abort*. Bus fault juga bisa terjadi saat proses penyimpanan (PUSH) atau pengambilan (POP) data ke dan dari memori stack di awal dan akhir interupsi. Kejadian ini juga dinamakan dengan *stacking* dan *unstacking error*. Ketika eksepsi ini terjadi dan fungsi penanganannya diaktifkan, maka prosesor akan mengerjakan fungsi tersebut, tentunya dengan melihat level prioritasnya. Sedangkan jika tidak diaktifkan, prosesor akan membangkitkan eksepsi Hard fault.

Bus fault diaktifkan dengan men-set bit BUSFAULTENA di register SHCSR. Sedangkan statusnya bisa dilihat di register BFSR. Eksepsi Bus fault saat akses data dikategorikan menjadi dua, yaitu *precise* dan

imprecise. Eksepsi Bus fault *Imprecise* terjadi jika eksepsi disebabkan oleh suatu operasi yang dikerjakan (seperti penulisan terbufer) beberapa siklus clock sebelumnya. Sedangkan Bus fault *precise* disebabkan oleh operasi terakhir yang sudah dikerjakan, misalnya pembacaan memori di sebut *precise* karena instruksi tidak bisa selesai sampai menerima data.

Tabel 2.7 Register BFSR

Bit	Nama	Sifat	Nilai Reset	Keterangan
7	BFARVALID		0	Menunjukan register BFAR valid, berisi alamat memori yang menyebabkan Bus fault
6:5	-	-	-	-
4	STKERR	Baca/Tulis	0	Stacking Error
3	UNSTKERR	Baca/Tulis	0	Unstacking Error
2	IMPRECISERR	Baca/Tulis	0	Pelanggaran saat akses data <i>Imprecise</i>
1	PRECISERR	Baca/Tulis	0	Pelanggaran akses data <i>Precise</i>
0	IBUSERR	Baca/Tulis	0	Pelanggaran akses instruksi

2.5.4 USAGE FAULT

Usage fault berkaitan dengan eksepsi yang terjadi karena eksekusi sebuah instruksi. Beberapa hal penyebabnya sebagai berikut:

1. Ketika prosesor menjalankan instruksi yang tidak terdefinisi.
2. Ketika prosesor mencoba menjalankan instruksi untuk coprosesor, padahal ARM Cortex-M3 tidak mendukung coprosesor.
3. Ketika menjalankan instruksi yang mencoba mengganti ke instruksi ARM (*ARM State*), padahal ARM Cortex-M3 tidak mendukung instruksi ARM.
4. Kesalahan di alamat kembali setelah terjadinya interupsi, dalam hal ini register Link (R14) berisi nilai yang tidak benar.
5. Kesalahan saat mengakses memori tak rata (*unaligned*).

Seperti eksepsi-eksepsi sebelumnya, eksepsi Usage fault juga harus diaktifkan melalui bit USGFAULTENA di register SHCSR. Jika tidak, maka ketika terjadi Usage fault, prosesor akan membangkitkan eksepsi Hard fault. Status dari Usage fault bisa dilihat di register UFSR (*Usage*

Fault Status Register) yang bisa diakses di alamat 0xE000ED28 secara 32 bit (word) atau 0xE000ED2A secara 16 bit (*half word*).

Tabel 2.8 Register UFSR

Bit	Nama	Sifat	Nilai Reset	Keterangan
9	DIVBYZERO	Baca/Tulis	0	Usage fault disebabkan oleh operasi pembagian oleh bilangan nol.
8	UNALIGNED	Baca/Tulis	0	Usage fault disebabkan oleh akses memori tak rata
7:4	-	-	-	-
3	NOCP	Baca/Tulis	0	Eksekusi instruksi coprosesor
2	INVPC	Baca/Tulis	0	Usage fault disebabkan oleh kesalahan di alamat kembali setelah interupsi
1	INVSTATE	Baca/Tulis	0	Mencoba mengeksekusi instruksi ARM
0	UNDEFINSTR	Baca/Tulis	0	Mengeksekusi instruksi yang tidak dikenali

2.5.5 SVC, PENDSV DAN SYSTICK

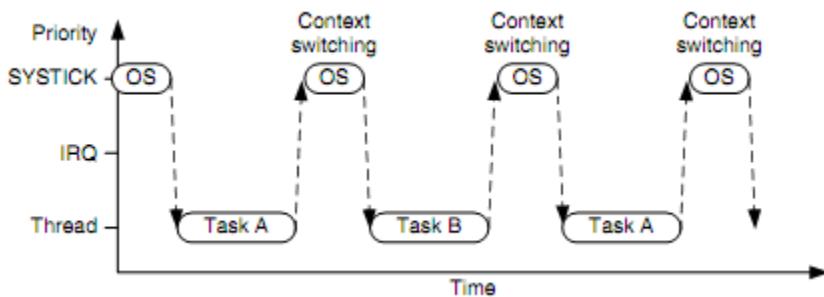
Selain eksepsi yang dipicu oleh adanya kesalahan (*fault*), ARM Cortex-M3 mengenal juga eksepsi yang dibangkitkan secara software, terutama ketika ARM Cortex menjalankan sebuah RTOS. Ada 3 jenis eksepsi ini, yaitu:

1. SVC (*Supervisor Call*)
2. PendSV (*Pendable Service Call*)
3. SysTick

SVC (*Supervisor Call*) dan PendSV (*Pendable Service Call*) merupakan eksepsi yang dibangkitkan secara software. SVC dibangkitkan dengan memanggil instruksi SVC, sedangkan PendSV dibangkitkan dengan menet bit PENDSVSET di register ICSR (*Interrupt Control and State Register*). Systick merupakan eksepsi yang dibangkitkan ketika timer SysTick (*System Tick*), yang merupakan timer 24 bit, menghitung mundur mencapai nol. Ketiga eksepsi ini digunakan menjalankan sebuah kernel sistem operasi (RTOS). Dalam sistem berbasis RTOS, setiap task akan dijalankan secara bergantian dalam waktu tertentu, dan ini diatur oleh kernel OS atau *scheduler* sehingga sistem terlihat seperti multi-tasking. Pergantian antar task dinamakan dengan pergantian konteks (*context*

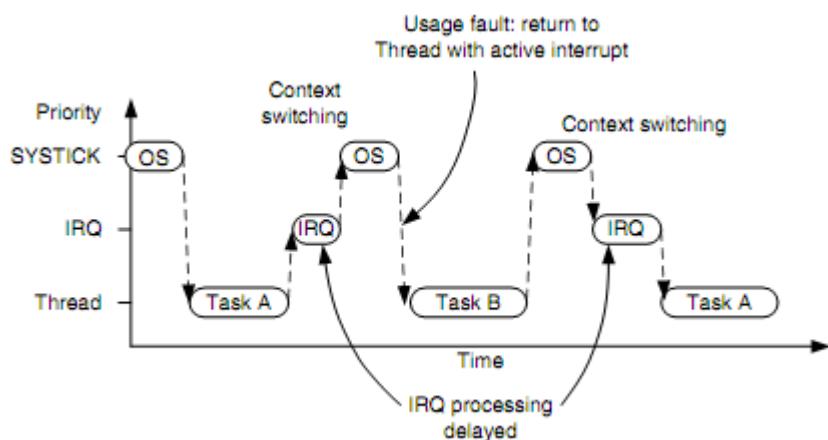
switching), dan ketiga eksepsi ini digunakan dalam pergantian konteks tersebut.

Eksepsi SVC juga bisa digunakan apabila dalam sebuah sistem berbasis RTOS, setiap task atau fungsi tidak diijinkan mengakses hardware atau periperal internal secara langsung, tetapi harus melalui kernel (OS), kernel-lah yang kemudian menghubungkannya ke hardware melalui fungsi ke hardware tersebut (*device driver*). Hal ini bisa dilakukan ketika akan mengakses hardware tersebut, task akan memanggil instruksi SVC dan fungsi penanganan yang ada dalam RTOS (*SVC handler*) kemudian menghubungkannya ke hardware.



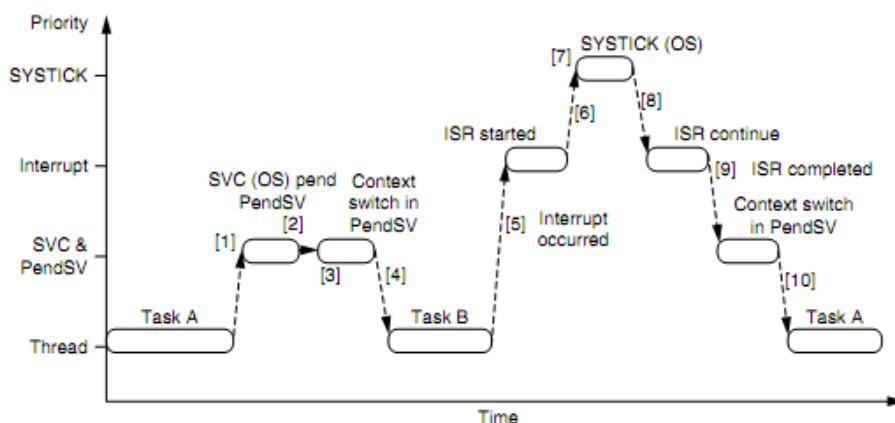
Gambar 2.19 Pergantian Konteks dalam RTOS

Gambar 2.19 di atas menggambarkan bagaimana sebuah pergantian konteks atau pergantian task dari Task A ke Task B dan seterusnya dilakukan setiap terjadi eksepsi SYSTICK. Masalah akan terjadi ketika akan terjadi pergantian konteks, terjadi interupsi dari periperal internal, misalnya dari port serial. Pelayanan terhadap permintaan interupsi tersebut akan tertunda karena didahului oleh pergantian konteks tersebut. Atau bisa saja interupsi yang didahului dan pergantian konteks yang ditunda, tetapi ARM Cortex-M3 akan membangkitkan eksepsi Usage fault ketika OS berusaha berganti ke mode thread ketika sebuah interupsi sedang aktif.



Gambar 2.20 Interupsi terjadi sesaat sebelum pergantian konteks

Permasalahan di atas bisa diatasi dengan cara OS akan melakukan pergantian konteks ketika tidak sedang mengeksekusi pelayanan interupsi (ISR). Namun akan berakibat adanya waktu tunda yang panjang saat pergantian task terutama ketika interupsi sering terjadi berdekatan dengan saat pergantian task. PendSV bisa mengatasi hal ini. PendSV, sesuai namanya *Pendable Service Call*, akan menunda permintaan pergantian konteks sampai semua fungsi pelayanan interupsi selesai dikerjakan.



Gambar 2.21 Pergantian Konteks dengan PendSV

Sebuah interupsi terjadi sesaat sebelum pergantian konteks. Pelayanan interupsi tersebut akan ditunda terlebih dahulu untuk mengerjakan eksepsi SYSTICK. Dengan instruksi PendSV, setelah SYSTICK selesai OS

tidak langsung mengerjakan pergantian konteks, dari Task B ke Task A, tetapi melanjutkan dulu pelayanan interupsi yang tadi tertunda. Setelah interupsi selesai dilayani, pergantian konteks sesungguhnya akan dilakukan, dan ini tidak akan membangkitkan eksepsi Usage fault. Oleh karena itu, PendSV lebih umum digunakan dalam pergantian konteks sebuah OS.

2.5.6 TABEL VEKTOR

Ketika terjadi sebuah eksepsi, prosesor ARM Cortex-M3 memerlukan informasi tentang awal dari fungsi pelayanan eksepsi tersebut. Informasi ini disimpan dalam sebuah tabel vektor di dalam memori. Setelah reset tabel vektor berada di alamat offset 0x0000 yang merupakan nilai awal dari penunjuk stack. Alamat selanjutnya adalah alamat awal dari fungsi pelayanan eksepsi atau interupsi eksternal (peripheral) yang diatur dan diurutkan menurut nomor eksepsi dikalikan 4, seperti ditunjukkan oleh gambar 2.22. Di namakan alamat offset karena alamat sebenarnya mengacu kepada alamat dari memori flash atau RAM di mana kode atau program disimpan, tentunya area memori yang merupakan area yang bisa dieksekusi (*boot code*).

Tabel vektor bisa dipindahkan ke area atau alamat lain saat program berjalan. Hal ini bisa dilakukan dengan mengubah isi register VTOR (*Vector Table Offset Register*). Offset alamat harus disesuaikan dengan ukuran vektor tabel (jumlah eksepsi), dan harus ditambah sehingga jumlah eksepsinya menjadi bilangan dalam deret bilangan pangkat 2. Misalnya sebuah prosesor mempunyai 32 sumber interupsi dari periperal (IRQ), maka jumlah eksepsi adalah $32+16$ (eksepsi sistem dan fault) = 48. Bilangan pangkat 2 terdekat yang lebih besar dari pada 48 adalah 64, maka perpanjangan jumlah eksepsinya menjadi 64. Dengan ketentuan tiap offset harus ada jarak 4 byte, maka ukuran tabel menjadi $64 \times 4 = 256$ byte (0x100). Sehingga tabel vektor bisa diprogram di alamat dengan kelipatan 0x100 yaitu 0x0, 0x100, 0x200 dan seterusnya.

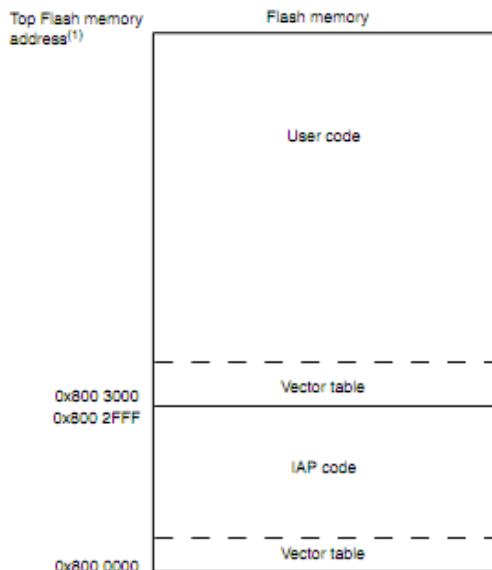
Perubahan tabel vektor ini berguna ketika mengembangkan aplikasi *boot loader* atau *In-System Application Programming* (IAP). Umumnya ada 3 cara untuk memprogram memori flash mikrokontroler, melalui jalur *In-System Programming* (ISP) lewat port debug (JTAG atau SWD), melalui boot loader yang sudah ditanam dari manufaktur (biasanya melalui port serial), atau boot loader yang dibuat sendiri (IAP). Dalam IAP, program

bisa dimasukan melalui jalur komunikasi yang tersedia, paralel (GPIO), uart, I2C, SPI, USB, Ethernet dan lain-lain.

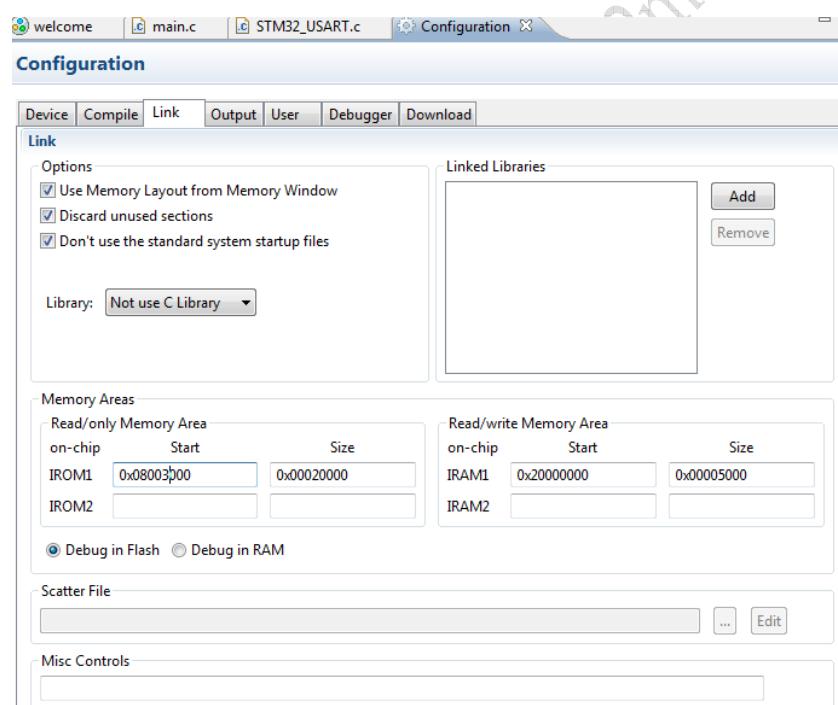
Exception number	IRQ number	Offset	Vector
83	67	0x014C	IRQ67
:	:	:	:
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

Gambar 2.22 Tabel Vektor

Dalam IAP, area memori flash akan dibagi menjadi 2 yaitu program IAP dan program aplikasi. Baik program IAP maupun program aplikasi mempunyai tabel vektor masing-masing. Tabel vektor IAP tetap di alamat offset 0x0, sedangkan vektor aplikasi disesuaikan dengan besarnya ukuran program IAP dan ketentuan di atas. Pergantian tabel vektor, mengubah register VTOR harus dilakukan dalam program aplikasi dan saat proses *build* alamat awal memori flash juga harus disesuaikan.



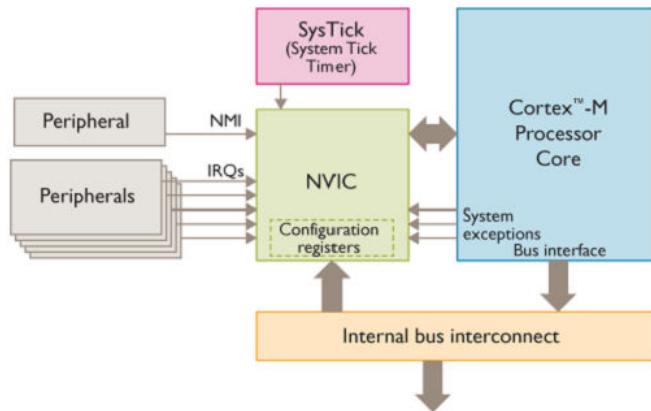
Gambar 2.23 Pemetaan Memori Flash IAP



Gambar 2.24 Pengaturan Alamat Awal Flash di CoIDE

2.6 KENDALI INTERUPSI

Keluarga ARM Cortex mempunyai sistem kendali interupsi yang dinamakan dengan NVIC (*Nested Vectored Interrupt Controller*). NVIC ini merupakan bagian dari core Cortex-M3, sehingga semua prosesor dengan core Cortex-M3 akan mempunyai struktur interupsi yang sama. NVIC dirancang untuk bisa menangani sampai 240 interupsi eksternal dari periperal (IRQ), selain itu NVIC juga terhubung dengan eksepsi-eksepsi yang dibangkitkan oleh core ARM Cortex sendiri, seperti yang telah dijelaskan sebelumnya.



Gambar 2.25 Struktur Interupsi ARM Cortex-M3

Sesuai dengan namanya, NVIC mendukung interupsi bersarang (*nested interrupt*). Dalam interupsi bersarang, prosesor memungkinkan untuk melayani interupsi yang baru walaupun pada saat itu sedang melayani interupsi lain, tentunya dengan melihat level prioritas dari masing-masing interupsi. Sehingga ketika terjadi sebuah interupsi, kendali interupsi (NVIC) akan melihat dan membandingkan dengan level prioritas dari interupsi yang sedang dilayani. Jika prioritas interupsi yang baru lebih tinggi dari prioritas interupsi yang sedang dikerjakan, maka interupsi yang sedang dilayani akan ditunda terlebih dahulu, dan prosesor akan melayani interupsi yang baru tersebut. Interupsi yang ditunda tersebut akan berstatus aktif dan tertunda (*active and pending*). Sebaliknya jika prioritas interupsi baru tersebut lebih rendah dari pada prioritas dari interupsi yang sedang dikerjakan, interupsi baru tersebut akan ditunda dan berstatus tertunda (*pending*).

Dengan skenario interupsi bersarang seperti di atas, sebuah interupsi akan mempunyai salah satu dari status di bawah ini:

1. Interupsi tidak aktif (*inactive*), jika interupsi atau eksepsi tidak aktif dan tidak ditunda
2. Interupsi ditunda (*pending*), jika interupsi pelayanannya ditunda karena prosesor sedang melayani interupsi dengan prioritas yang lebih tinggi.
3. Interupsi sedang aktif, jika interupsi masih dilayani oleh prosesor
4. Interupsi aktif dan tertunda, jika interupsi dengan dilayani tetapi kemudian ditunda karena ada interupsi dengan prioritas yang lebih tinggi.

NVIC juga dirancang agar ARM Cortex-M3 mempunyai waktu *latency* yang rendah. Waktu latency adalah waktu yang diperlukan oleh prosesor untuk mengeksekusi instruksi pertama dari fungsi layanan interupsi (ISR) atau fungsi penanganan eksepsi. ARM Cortex-M3 mempunyai waktu latency sebesar 12 siklus, sehingga sesuai untuk dipakai di aplikasi-aplikasi *real time*.

2.6.1 REGISTER-REGISTER NVIC

NVIC dikendalikan oleh beberapa register:

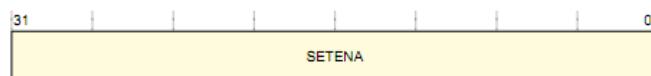
1. Register untuk mengaktifkan dan mematikan interupsi (ISER dan ICER)
2. Register penundaan interupsi (ISPR dan ICPR)
3. Register yang menyimpan status aktif tidaknya interupsi (IABR)
4. Register untuk mengatur prioritas interupsi (IPR)

Selain itu proses interupsi juga dipengaruhi oleh-oleh register-register khusus prosesor ARM Cortex-M3 (PRIMASK, FAULTMASK dan BASEPRI), tabel vektor, register interupsi software (STIR) dan pengelompokan prioritas.

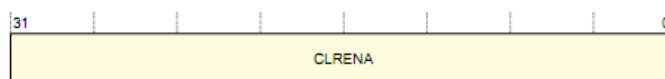
2.6.1.1 Register ISER dan ICER

Register ISER (*Interrupt Set Enable Register*) dan ICER (*Interrupt Clear Enable Register*) digunakan untuk mengaktifkan atau mematikan interupsi sebuah periperal. Interupsi diaktifkan dengan menset bit SETENA di register ISER, sedangkan untuk mematkannya dilakukan dengan menset bit CLRENA di register ICER. Kedua register ini merupakan register

32 bit, masing-masing bit mewakili satu sumber interupsi. Jika sebuah chip prosesor ARM Cortex-M3 memiliki lebih dari 32 interupsi eksternal, maka prosesor tersebut akan mempunyai register ISER dan ICER lebih dari satu. ARM Cortex-M3 maksimal akan mempunyai 8 buah register ISER dan ICER karena ARM Cortex-M3 bisa menangani IRQ sampai 240 IRQ. Hasil pembacaan register ini akan menunjukkan interupsi mana saja yang diaktifkan dan mana yang tidak.



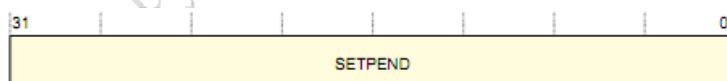
Gambar 2.26 Register ISER



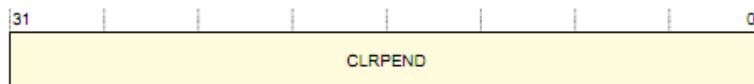
Gambar 2.27 Register ICER

2.6.1.2 Register ISPR dan ICPR

Register ISPR (*Interrupt Set Pending Register*) dan ICPR (*Interrupt Clear Pending Register*) digunakan untuk menunda sebuah interupsi, misalnya karena ada interupsi lain dengan prioritas interupsi yang lebih tinggi. Men-set bit SETPEND di register ISPR akan membuat interupsi yang diwakilinya menjadi tertunda (*pending*) dan men-set bit CLRPEND di register ICPR akan membatalkan status penundaan sebuah interupsi. Kedua register ini juga merupakan register 32 bit, dan setiap bit mewakili sebuah interupsi. Prosesor Cortex-M3 bisa mempunyai lebih dari satu register ISPR dan ICPR sesuai dengan jumlah interupsi eksternal yang dimilikinya. ARM Cortex-M3 maksimal akan mempunyai 8 buah register ISPR dan ICPR karena ARM Cortex-M3 bisa menangani IRQ sampai 240 IRQ. Hasil pembacaan register ini juga akan menunjukkan interupsi mana saja yang sedang ditunda.



Gambar 2.28 Register ISPR

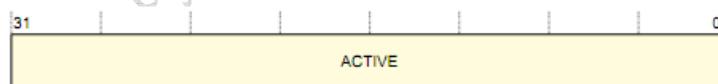


Gambar 2.29 Register ICPR

Register ISPR dan ICPR bisa dimodifikasi secara software, sehingga bisa digunakan untuk membatalkan sebuah interupsi atau membangkitkan interupsi secara software. Interupsi yang ditunda karena ada interupsi lain dengan prioritas lebih tinggi, bisa dibatalkan dengan men-set bit CLRPEND di register ICPR. Begitu juga ketika software men-set bit SETPEND di register ISPR, walaupun sumber interupsinya tidak benar-benar terjadi, maka prosesor ARM Cortex akan tetap mengkondisikan interupsi tersebut sedang ditunda, sehingga ketika selesai melayani interupsi dengan prioritas lebih tinggi, prosesor akan mengerjakan fungsi layanan dari interupsi yang ditunda tersebut.

2.6.1.3 Register IABR

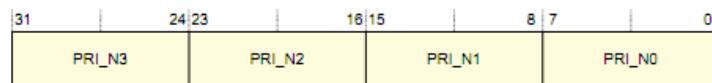
Register IABR (*Interrupt Active Bit Register*) menyimpan bit-bit status apakah interupsi sedang dieksekusi atau tidak. Saat terjadi interupsi, prosesor akan men-set bit ACTIVE dari register IABR sesuai dengan interupsi yang dilayani dan akan me-reset bit ACTIVE setelah fungsi layanan interupsi (ISR) yang bersangkutan selesai dieksekusi. Selama mengeksekusi ISR, ada kemungkinan terjadi interupsi dengan prioritas yang lebih tinggi, sehingga harus dilayani terlebih dahulu. Pada kondisi ini bit ACTIVE akan tetap di-set, tetapi untuk menandakan bahwa interupsinya sedang ditunda, prosesor akan men-set bit SETPEND di register ISPR. Sama dengan register sebelumnya, sebuah prosesor ARM Cortex-M3 bisa mempunyai register IABR lebih dari satu. ARM Cortex-M3 maksimal akan mempunyai 8 buah register IABR karena ARM Cortex-M3 bisa menangani IRQ sampai 240 IRQ. Register IABR bersifat hanya bisa dibaca.



Gambar 2.30 Register IABR

2.6.1.4 Register IPR

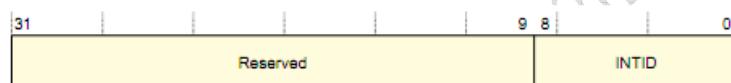
Register IPR (*Interrupt Priority Register*) merupakan register yang digunakan untuk mengatur prioritas sebuah interupsi. Register ini bisa diakses secara byte sebagai register IP, sehingga dalam 1 register IPR akan ada 4 register IP. Register IP ini menyimpan prioritas 4 bit untuk setiap interupsi, seperti yang akan dijelaskan.



Gambar 2.31 Register IPR

2.6.1.5 Register STIR

Register STIR (*Software Trigger Interrupt Register*), sesuai dengan namanya digunakan untuk membangkitkan interupsi secara software. Interupsi software dibangkitkan dengan menuliskan identitas (ID) interupsi ke bit-bit NTID (8 bit) di register STIR (0 – 239). Misalnya ketika 0x03 dimasukan ke register STIR, maka akan dibangkitkan interupsi dari IRQ3.



Gambar 2.32 Register STIR

Register STIR hanya bisa diakses secara privileged, jika mode user (unprivileged) ingin mengakses register STIR, maka bit USERSETMPEND di register SCR (*System Control Register*) harus di-set terlebih dahulu. Tetapi untuk men-set bit ini dibutuhkan juga akses privileged.

Selain register-register di atas, register PRIMASK, FAULTMASK dan BASEPRI yang merupakan register khusus dari prosesor ARM Cortex-M3 juga bisa digunakan untuk mengendalikan interupsi. Sub-bab sebelumnya telah membahas register-register khusus ini secara singkat.

Register PRIMASK bisa digunakan untuk memblok semua interupsi dan eksepsi kecuali NMI dan Hard fault. Men-set bit PRIMASK (register PRIMASK hanya dipakai 1 bit) akan membuat prioritas interupsi menjadi 0, nilai prioritas tertinggi yang bisa diprogram. Sehingga interupsi atau eksepsi dengan prioritas 0 atau lebih akan diblok, kecuali NMI dan Hard fault yang mempunyai nilai prioritas interupsi -2 dan -1. Register

PRIMASK berguna untuk memblok sementara interupsi-interupsi ketika sedang mengerjakan fungsi kritis yang tidak bisa diganggu oleh interupsi yang lain.

Register FAULTMASK pada dasarnya hampir sama dengan PRIMASK, bedanya FAULTMASK akan membuat prioritas interupsi menjadi -1, sehingga semua interupsi yang nilai prioritasnya sama atau lebih besar -1 akan diblok. Ini artinya semua interupsi, kecuali NMI akan diblok.

Register BASEPRI digunakan juga untuk memblok interupsi berdasarkan nilai prioritasnya, dengan BASEPRI nilai prioritas interupsi yang bisa diblok bisa diprogram memasukan nilai prioritas tersebut ke register BASEPRI.

2.6.2 PRIORITY INTERUPSI

Dengan sistem interupsi bersarang ketika terjadi sebuah interupsi atau eksepsi, prosesor akan selalu membaca apakah prioritas interupsi bisa menunda atau mendahului (*preempt*) interupsi yang sedang dieksekusi. Di prosesor ARM Cortex-M3, interupsi dengan nilai prioritas lebih kecil akan mempunyai prioritas lebih tinggi dan sebaliknya. Prioritas ini ada yang dibuat tetap (*fix*) dan ada yang bisa diprogram. Prioritas interupsi yang tetap, seperti ditunjukkan oleh tabel 2.4, adalah reset, NMI dan Hard fault. Reset adalah eksepsi tertinggi dengan nilai prioritas -3.

Prioritas interupsi ARM Cortex-M3 diatur melalui register IPR (32 bit) yang terdiri dari 4 register IP bila diakses sebagai register 8 bit, sehingga satu register IPR mengatur 4 prioritas interupsi. Dengan register prioritas 8 bit, ARM Cortex-M3 akan mempunyai 3 level prioritas interupsi (Reset, NMI dan Hard fault) dan 256 level prioritas interupsi yang dapat diprogram (8 bit). Bandingkan dengan mikrokontroler keluarga 8051 yang hanya mempunyai dua level prioritas interupsi (1 bit). Walaupun demikian setiap pabrik silikon yang memproduksi prosesor ARM Cortex-M3 akan menerapkan prioritas interupsi yang berbeda-beda. Bisa menerapkan 8 level prioritas (3 bit), 16, 32 dan seterusnya. Level minimal adalah 8 level prioritas (3 bit). Lebih banyak level prioritas interupsi yang diterapkan, maka lebih banyak jumlah gerbang-gerbang digital yang harus dibuat di dalam core ARM Cortex-M3, artinya akan lebih banyak mengkonsumsi daya. Selain itu juga, akan berdampak ke harga jual prosesor.

ARM Cortex-M3 dengan level prioritas kurang dari 8 bit, akan menggunakan bit-bit MSB (*Most Significant Bit*) dari register IP dan mengabaikan bit-bit LSB (*Least Significant Bit*). Gambar 2.33 menunjukan contoh implementasi prioritas interupsi dengan 3 bit. Oleh karena bit ke-4 sampai bit ke-0 tidak dipakai, dan akan selalu terbaca 0, maka level prioritas interupsi akan bernilai 0x00 (sebagai prioritas tertinggi), 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0, dan 0xE0 (sebagai prioritas terendah).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented	Not implemented						

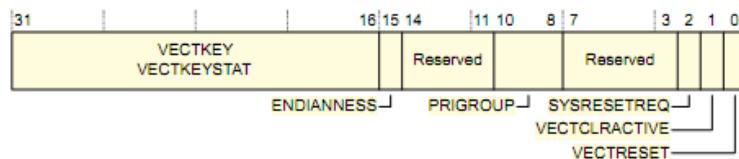
Gambar 2.33 Prioritas Interupsi dengan 3 Bit

Alasan kenapa menggunakan bit-bit MSB dari pada bit-bit LSB adalah untuk memudahkan mem-porting software dari satu prosesor Cortex-M3 ke prosesor Cortex-M3 yang lain. Dengan cara ini, sebuah software yang dikembangkan untuk Cortex-M3 dengan prioritas 4 bit bisa langsung dijalankan di prosesor Cortex-M3 dengan prioritas 3 bit. Jika yang digunakan bit-bit LSB, maka akan terjadi pembalikan prioritas interupsi ketika sebuah program dijalankan di prosesor yang mempunyai beda level interupsinya. Sebagai contoh prosesor dengan level interupsi 4 bit, IRQ0 mempunyai prioritas 0x09 dan IRQ1 mempunyai prioritas 0x07. Dari sini diketahui bahwa IRQ1 akan mempunyai prioritas lebih tinggi. Jika digunakan di prosesor dengan level interupsi 3 bit, dengan membuang bit ke-3, maka IRQ0 akan menjadi 0x01 dan IRQ1 akan tetap 0x07, tetapi IRQ0 sekarang menjadi lebih tinggi prioritasnya.

Level prioritas interupsi dibagi menjadi 2 bagian, yaitu level prioritas *preempt* dan level *subpriority*. Level prioritas *preempt* menentukan apakah sebuah interupsi bisa mendahului atau menunda interupsi yang sekarang sedang dikerjakan oleh prosesor (interupsi bersarang), di mana level prioritas preempt yang lebih kecil akan mempunya nilai prioritas yang lebih besar. Sedangkan level subprioritas digunakan ketika ada 2 interupsi dengan level prioritas preempt yang sama terjadi secara bersamaan di mana interupsi dengan nilai sub-prioritas yang lebih kecil akan didahulukan.

Bit-bit prioritas preempt dan sub-prioritas diatur dan bisa dikelompokan ke dalam kelompok prioritas (*priority grouping*) melalui bit PRIGROUP di register AIRCR (*Application Interrupt and Reset Control Register*). Bit

PRIGROUP ini terdiri dari 3 bit, bit ke-8 sampai bit ke-10, sehingga ARM Cortex-M3 mendukung sampai 8 level pengelompokan group prioritas (0 – 7). Hubungan antara level kelompok prioritas dengan level preempt dan sub-prioritas ditunjukan oleh tabel 2.9.



Gambar 2.34 Register AIRCR

Tabel 2.9 Kelompok Prioritas dan Register IP

Kelompok Prioritas	Level Preempt	Sub-Prioritas
0	Bit[7:1]	Bit[0]
1	Bit[7:2]	Bit[1:0]
2	Bit[7:3]	Bit[2:0]
3	Bit[7:4]	Bit[3:0]
4	Bit[7:5]	Bit[4:0]
5	Bit[7:6]	Bit[5:0]
6	Bit[7]	Bit[6:0]
7	Tidak ada	Bit[7:0]

Dari tabel terlihat, walaupun ARM Cortex-M3 menggunakan level prioritas interupsi 8 bit, tetapi karena adanya kelompok prioritas, hanya bisa mendukung 128 level preempt, yaitu ketika bit kelompok prioritas (PRIGROUP) bernilai 0. Dan ketika kelompok prioritas bernilai 7, maka semua interupsi/eksepsi yang prioritasnya bisa diprogram akan berada di sub-prioritas yang sama, dan tidak ada interupsi yang bisa menunda atau mendahului interupsi yang sedang dikerjakan, karena tidak ada level preempt.

Jika sebuah prosesor ARM Cortex-M3 menggunakan level prioritas 3 bit, dan kelompok prioritas di set ke level 5, maka konfigurasi register IP akan seperti gambar 2.35. Dari tabel di atas, jika kelompok prioritas di level 5, maka prioritas preempt akan berada di bit ke-6 dan ke-7, sedangkan bit sub-prioritas akan berada di bit ke-0 sampai bit ke-5, namun karena

hanya 3 bit yang dipakai, maka sub-prioritas hanya akan memakai bit ke-5. Sehingga prosesor akan mempunyai 4 level prioritas preempt (bit ke-6 dan bit ke-7) dan 2 level sub-prioritas (bit ke-5).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Preempt priority	Sub-priority	Not implemented					

Gambar 2.35 Kelompok Prioritas 7 dan Level Prioritas 3 bit

Contoh lain ketika sebuah prosesor menggunakan konfigurasi prioritas 8 bit dengan kelompok prioritas di level 0, maka prosesor akan bisa mendukung sampai dengan 128 level prioritas dan 2 level sub-prioritas.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Preempt priority						Subpriority	

Gambar 2.36 Kelompok Prioritas dan Level Prioritas 8 bit

Secara sederhana aturan prioritas ARM Cortex-M3 adalah sebagai berikut:

1. Ketika terjadi sebuah interupsi atau eksepsi, maka level prioritas preempt dari interupsi tersebut akan dibandingkan dengan prioritas preempt dari interupsi yang sedang dieksekusi, jika prioritasnya lebih tinggi, interupsi yang sedang dieksekusi akan ditunda dan prosesor akan melayani interupsi yang baru. Jika prioritasnya lebih rendah, interupsi yang baru tersebut akan ditunda.
2. Jika terjadi 2 atau lebih interupsi dengan level preempt yang sama, maka prosesor akan melihat level sub-prioritas. Interupsi dengan level sub-prioritas lebih tinggi (nilai sub-prioritas lebih rendah) akan didahulukan.
3. Jika level sub-prioritasnya sama, maka prosesor akan melihat nomor eksepsi/interupsi tersebut, nomor eksepsi lebih rendah akan didahulukan. IRQ0 akan mempunyai prioritas lebih tinggi dari IRQ1.

2.6.3 KARAKTERISTIK INTERUPSI

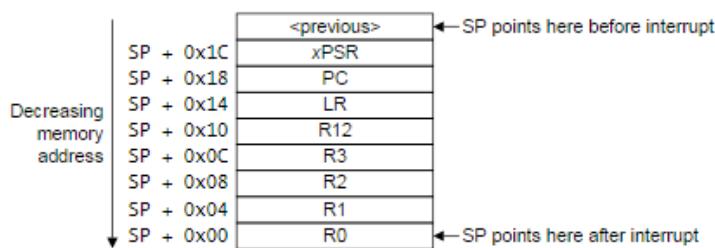
Selain mendukung interupsi bersarang dengan berbagai pengaturan prioritas, interupsi ARM Cortex-M3 dirancang agar bersifat deterministik,

artinya waktu latency setiap terjadi interupsi adalah tetap dan bisa ditentukan. Sistem interupsi ARM7 dan ARM9, tidak bersifat deterministik, waktu yang diperlukan untuk menghentikan sebuah instruksi yang sedang dikerjakan ketika interupsi terjadi bervariasi. Pada beberapa aplikasi mungkin hal ini tidak akan bermasalah, tapi pada sistem yang membutuhkan *real time* bisa jadi masalah besar. Beberapa teknik untuk memperkecil waktu latency diterapkan di prosesor ARM Cortex-M3.

2.6.3.1 Proses Saat Terjadi dan Setelah Interupsi

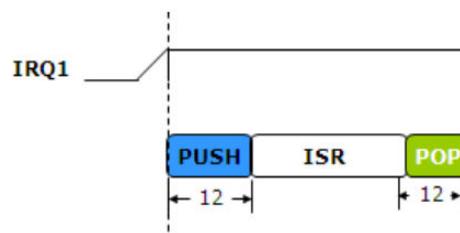
Ketika terjadi interupsi atau eksepsi, prosesor akan memanggil fungsi layanan interupsi sesuai dengan alamat vektor dari interupsi yang bersangkutan. Alamat program yang sedang dieksekusi secara otomatis akan disimpan ke memori stack, sehingga program bisa kembali ke alamat tersebut setelah fungsi interupsi selesai dikerjakan. Di fungsi layanan interupsi pastinya akan menggunakan register-register yang juga sedang dipakai di program utama dan hal ini akan merusak jalannya program utama ketika interupsi telah selesai dan program kembali ke program utama. Oleh karena itu register-register tersebut harus disimpan dulu datanya ke memori stack. Setelah interupsi selesai, data tersebut diambil kembali dan disimpan ke register yang sesuai sehingga data di register akan tetap seperti ketika interupsi belum terjadi.

Oleh karena itu, fungsi layanan interupsi biasanya diawali dengan proses penyimpanan ke memori stack (*push*) dan diakhiri dengan proses pengambilan kembali data dari memori stack (*pop*). Hal ini tentu akan ada tambahan waktu sebelum fungsi layanan interupsi yang sebenarnya bisa dieksekusi dan juga tambahan waktu setelah fungsi interupsi selesai dikerjakan. Prosesor ARM Cortex-M3 mengatasi hal ini dengan menggunakan *microcode* sehingga proses penumpukan (*stacking*) atau penyimpanan data dan pengambilan kembali (*unstacking*) register-register berlangsung secara otomatis, tidak perlu dituliskan di program layanan interupsi. Microcode bisa dikatakan sebagai program yang ditanamkan ketika prosesor dipabrikasi, dan bisa dijalankan untuk kepentingan tertentu. Register-register yang disimpan di stack ini meliputi R0 – R3, R12, LR, PC dan PSR, dinamakan sebagai *stack frame*. Lokasi memori stack yang digunakan bisa stack proses (PSP) maupun stack utama (MSP) tergantung saat itu program sedang menggunakan yang mana.



Gambar 2.37 Register-Register yang Disimpan ke Memori Stack

Ketika interupsi terjadi, microcode langsung melakukan penyimpanan register-register ke memori stack melalui bus data. Bersamaan dengan itu, alamat dari ISR yang bersangkutan dibaca melalui bus instruksi. Karena prosesor menggunakan arsitektur Harvard, maka kedua hal itu bisa dilakukan secara bersamaan. Sehingga proses penyimpanan dan pengambilan kembali data-data register hanya membutuhkan waktu 12 siklus.

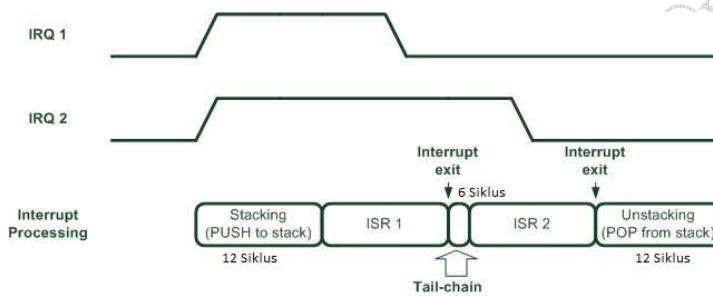


Gambar 2.38 Waktu Latency hanya 12 Siklus

Register-register yang disimpan oleh microcode mengacu kepada *Procedure Call Standard for the ARM Architecture* yang dikeluarkan oleh ARM Ltd. Menurut standar tersebut, register-register R0 – R3 adalah register-register yang digunakan untuk menyimpan parameter sebuah fungsi dalam bahasa C. Register R12 juga sering digunakan ketika sebuah fungsi dipanggil. Dengan disimpannya register-register tersebut, maka ISR aman untuk ditulis dalam bahasa C. Selain itu, untuk kembali dari fungsi interupsi, ARM Cortex tidak memerlukan instruksi khusus. Tidak seperti mikrokontroler keluarga 8051 yang menggunakan instruksi RETI (*Return from Interrupt*), ARM Cortex menggunakan instruksi kembali seperti normalnya kembali dari sebuah fungsi yang dipanggil. Sehingga fungsi ISR prosesor ARM Cortex bisa seluruhnya ditulis dalam bahasa C.

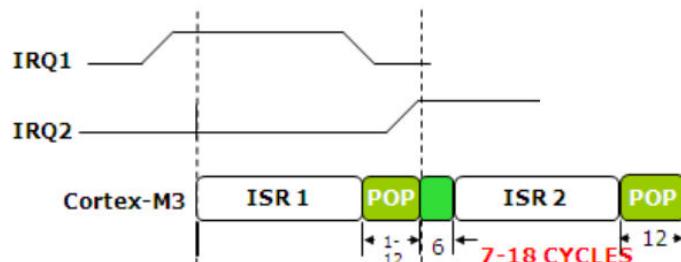
2.6.3.2 Tail-Chaining

Ketika sebuah interupsi terjadi sedangkan prosesor sedang melayani interupsi dengan prioritas yang sama atau lebih tinggi, maka interupsi tersebut akan ditunda. Begitu juga ketika terjadi 2 interupsi secara bersamaan, maka interupsi dengan prioritas lebih rendah akan ditunda dan prosesor akan mengerjakan interupsi dengan prioritas lebih tinggi. Setelah prosesor selesai melayani interupsi tersebut, sebelum microcode melakukan operasi pengambilan data dari memori stack (operasi pop), prosesor akan mengecek apakah saat itu ada interupsi yang tertunda atau tidak, jika ada maka prosesor tidak akan melakukan operasi pop, tetapi langsung mengambil alamat vektor interupsi tertunda tersebut dan mengerjakan fungsi ISRnya. Proses ini hanya membutuhkan 6 siklus. Teknik ini dinamakan dengan *tail-chaining*, suatu proses untuk mempercepat pelayanan interupsi.



Gambar 2.39 Tail-Chaining

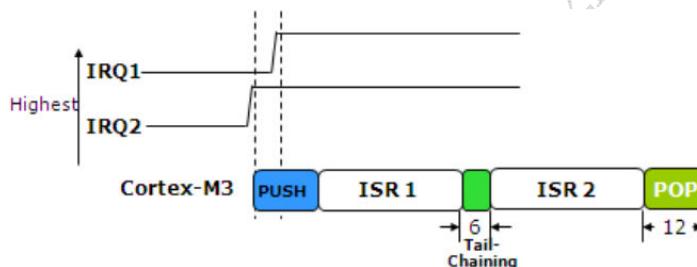
Jika terjadinya interupsi baru ketika prosesor sedang mengakhiri interupsi yang sekarang (operasi pop), maka prosesor bisa membatalkan operasi pop tersebut dan langsung mengambil alamat vektor tersebut. Tergantung sampai sejauh mana prosesor melakukan operasi pop, akan ada tambahan 1 siklus (jika operasi pop baru saja dimulai) sampai 12 siklus (jika operasi pop memang sudah selesai) ditambah 6 siklus proses pengambilan vektor interupsi, sebelum interupsi tersebut dikerjakan.



Gambar 2.40 Tail-Chaining Saat Interupsi Berakhir

2.6.3.3 Kedatangan yang Terlambat

Apabila saat ini prosesor sedang mengeksekusi dan sedang proses penumpukan data ke memori stack kemudian terjadi interupsi dengan prioritas yang lebih tinggi, tentunya prosesor akan mendahulukan interupsi dengan prioritas lebih tinggi tersebut. Proses penumpukan tetap dilanjutkan, prosesor akan mengerjakan interupsi dengan prioritas lebih tinggi terlebih dahulu baru kemudian interupsi yang lebih rendah dengan proses *tail-chaining*.



Gambar 2.41 IRQ1 datang Saat Proses Penumpukan

2.6.4 SETING INTERUPSI

Interupsi dalam sistem embedded merupakan salah satu teknik agar sistem embedded mempunyai tanggapan yang lebih cepat atau *real time* terhadap sebuah masukan atau input agar segera bisa diproses. Cara lainnya adalah dengan *polling*, namun dengan cara ini prosesor harus bekerja lebih karena prosesor harus bertanya (*poll*) kepada semua input apakah input tersebut mempunyai data untuk diproses atau tidak. Respon pun akan lebih lambat, karena sebuah masukan yang membutuhkan pemrosesan data harus menunggu giliran untuk ditanya. Dengan menggunakan interupsi, maka pada saat tidak ada permintaan

dari sebuah input, maka prosesor bisa mengerjakan fungsi yang lain, atau malah dibuat di kondisi hemat daya (*power down*) terutama untuk sistem embedded yang menggunakan batere.

Ada beberapa tahapan sebelum interupsi bisa digunakan:

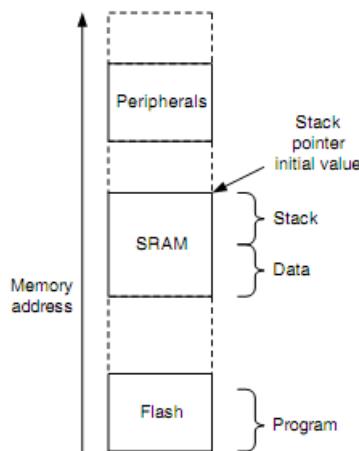
1. Pengaturan memori stack
2. Pengaturan tabel vektor
3. Pengaturan prioritas interupsi
4. Pengaktifkan interupsi dari periperal atau sistem eksepsi dari core Cortex-M3

2.6.4.1 Pengaturan Memori Stack

Seperti telah dijelaskan sebelumnya, ARM Cortex-M3 mempunyai 2 memori stack yaitu memori stack utama (MSP) dan memori stack proses (PSP). Untuk aplikasi-aplikasi sederhana, penggunaan hanya MSP mungkin sudah cukup. Jika aplikasi menggunakan RTOS, maka MSP dan PSP perlu digunakan agar data-data penting yang digunakan oleh kernel OS bisa dilindungi dari kemungkinan diubah oleh sebuah task. Tentu saja hal ini akan memerlukan ukuran memori stack yang cukup besar.

Dari gambar 2.37 terlihat, operasi stack ARM Cortex-M3 adalah mengurangi alamat memori, artinya setiap selesai penyimpanan data ke memori stack (*push*), alamat memori stack akan dikurangi (*descending stack*). Oleh karena itu, alamat awal memori stack biasanya ditempatkan di ujung atau alamat akhir dari RAM sehingga tidak tumpang tindih dengan data.

Setelah menentukan alamat awal memori stack, selanjutnya adalah menentukan besarnya memori stack. Besarnya memori stack tentunya tergantung dari aplikasi, ukuran terlalu besar tentunya akan membuat aplikasi boros memori, sedangkan ukuran memori stack terlalu kecil akan menyebabkan memori stack overflow dan bisa merusak data-data penting di memori utama. Pengaturan stack biasanya sudah dilakukan oleh IDE (*Integrated Development Environment*) melalui kode start up. Ketika akan mengubah pengaturan stack, maka dilakukan melalui kode start up ini.



Gambar 2.42 Contoh Penggunaan Memori Stack

2.6.4.2 Pengaturan Tabel Vektor

Seperti dijelaskan sebelumnya, pada dasarnya tidak perlu untuk mengubah atau merelokasi tabel vektor. Tabel vektor dikodekan ke dalam memori flash atau ROM dan diperlukan relokasi saat program berjalan.

Salah satu contoh aplikasi yang membutuhkan relokasi tabel vektor adalah IAP (sub-bab 2.5.6 telah membahas hal ini), di mana memori flash dibagi menjadi 2 bagian, yaitu bagian bootloader dan bagian aplikasi. Bootloader ini adalah program yang digunakan untuk menerima program yang nantinya akan ditulis ke bagian aplikasi. Program aplikasi yang harus merelokasi tabel vektornya.

2.6.4.3 Pengaturan Prioritas Interupsi

Setelah reset, semua interupsi yang prioritasnya bisa diprogram akan bernilai 0, artinya akan mempunyai prioritas yang sama. Pengubahan prioritas interupsi juga harus disesuaikan dengan aplikainya. Misalnya ketika sebuah fungsi tunda (*delay*) yang menggunakan interupsi timer Systick, jika fungsi ini akan digunakan di fungsi layanan sebuah interupsi, maka prioritas interupsi timer Systick harus dibuat lebih tinggi dibandingkan dengan prioritas interupsi yang memanggil fungsi *delay* tersebut.

2.6.4.4 Pengaktifan Interupsi

Pengaktifan sebuah interupsi dilakukan melalui register ISER, bit mana yang harus di-set tentunya tergantung interupsi apa saja yang akan diaktifkan. Bit-bit tersebut mengacu kepada kendali interupsi untuk masing-masing periperal. Dan tentunya mengacu kepada lembaran data (*datasheet*) dari masing-masing prosesor.

2.7 UNIT PROTEKSI MEMORI

Unit proteksi memori (MPU, *Memory Protection Unit*) merupakan unit pilihan prosesor ARM Cortex-M3, tidak semua prosesor ARM Cortex-M3 akan mempunyai MPU, tergantung dari pabrikan silikon yang bersangkutan. Dengan MPU memori prosesor ARM Cortex-M3 bisa dibagi-bagi menjadi region-region yang diproteksi, sehingga hanya bisa diakses melalui level akses tertentu dengan tujuan sebagai berikut:

1. Menjaga aplikasi user agar tidak merusak data yang dipakai oleh sistem operasi (RTOS).
2. Memisahkan data sehingga task pemroses data tidak bisa mengakses data yang lain.
3. Membuat beberapa region memori menjadi "hanya baca", sehingga menjaga data vital agar tidak bisa diubah dari yang seharusnya
4. Mendeteksi akses memori yang tidak diharapkan, misalnya memori stack rusak.

Setiap region memori yang sudah di-set oleh MPU mempunyai hak akses masing-masing. MPU bisa membuat sampai 8 region memori (area 0-7) dan region latar belakang (*background region*). Region latar belakang pada dasarnya sama dengan region memori yang lain, tetapi hanya bisa diakses di level privileged. MPU juga mengatur region - region tersebut mulai dari lokasi (alamat memori), ukuran, jenis akses, dan atribut memorinya. MPU mendukung:

1. Seting untuk masing-masing region memori
2. Region memori tumpang tindih (*overlapping region*)
3. Ekspor atribut memori ke sistem

Ketika sebuah region memori mengalami overlap, maka akses ke memori akan dipengaruhi oleh nomor region yang lebih tinggi, misalnya region 7 akan mempengaruhi region memori di bawahnya yang mengalami overlap ke region 7.

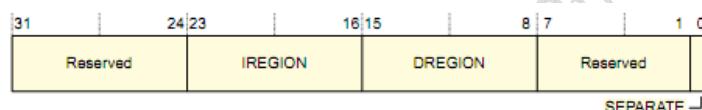
MPU dikendalikan melalui beberapa register:

1. Register type (MPU_TYPE)
2. Register kendali (MPU_CR)
3. Register nomer region (MPU_RNR)
4. Register Alamat dasar region (MPU_RBAR)
5. Register ukuran dan atribut region (MPU_RASR)

2.7.1 REGISTER-REGISTER MPU

2.7.1.1 Register TYPE

Register MPU_TYPE merupakan register yang menunjukkan apakah sebuah mikrokontroler ARM Cortex-M3 mempunyai unit MPU atau tidak. Register ini juga menunjukkan berapa region memori yang didukung oleh prosesor tersebut. Bit-bit IREGION (bit 23:16) merupakan bit-bit yang menunjukkan jumlah region instruksi yang didukung oleh MPU. Bit-bit ini akan selalu bernilai 0, karena arsitektur ARMv7-M menggunakan MPU yang terpadu antara data dan instruksi.



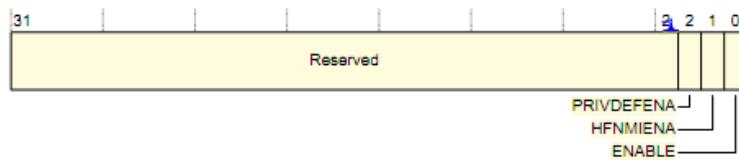
Gambar 2.43 Register MPU_TYPE

Bit-bit DREGION (bit 15:9), merupakan bit yang menunjukkan jumlah region data yang didukung oleh MPU. Bit-bit ini akan selalu bernilai 0x08 jika prosesor mempunyai MPU atau 0x00 jika prosesor tidak mempunyai MPU. Sedangkan bit SEPARATE merupakan bit yang menunjukkan apakah ada pemisahan antara pemetaan memori dan instruksi, dan akan selalu bernilai 0.

2.7.1.2 Register CR

Register MPU_CR merupakan register yang mengatur kerja MPU ini, register ini berfungsi untuk:

1. Mengaktifkan MPU
2. Mengaktifkan nilai awal dari region memori latar belakang
3. Mengaktifkan pemakaian MPU ketika terjadi Hard fault, NMI dan FAULTMASK.



Gambar 2.44 Register MPU_CR

Bit ENABLE, sesuai dengan namanya akan mengaktifkan fungsi MPU bila bit ini di-set. Bit HFNMENA digunakan untuk mengaktifkan MPU selama terjadi Hard fault, terjadi NMI dan penanganan FAULTMASK jika bit ini di-set. Dan bit PRIVDEFENA digunakan untuk mengaktifkan akses software privileged ke peta memori awal (default) sesuai dengan pemetaan memori ARM Cortex-M3.

2.7.1.3 Register RNR

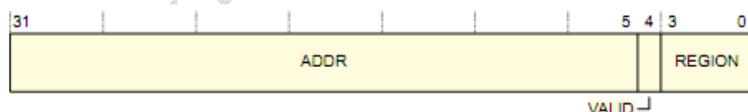
Register MPU_RNR (*Region Number Register*) merupakan register yang digunakan untuk memilih region memori mana yang akan di-set oleh register MPU_RBAR dan MPU_RASR. Software harus menentukan terlebih dahulu register RNR sebelum men-set register RBAR dan RASR.



Gambar 2.45 Register RNR

2.7.1.4 Register RBAR

Register MPU_RBAR (*Region Base Address Register*) merupakan register untuk menentukan alamat awal region memori di nomor region yang ditunjukkan oleh register RNR. Register ini terdiri dari bit-bit alamat (ADDE[31:N]), bit VALID dan bit-bit REGION[3:0].



Gambar 2.46 Register RBAR

Bit-bit ADDR[31:N] merupakan alamat awal region, di mana N tergantung ukuran region, yang ditunjukkan oleh bit-bit SIZE di register RASR. N dirumuskan sebagai

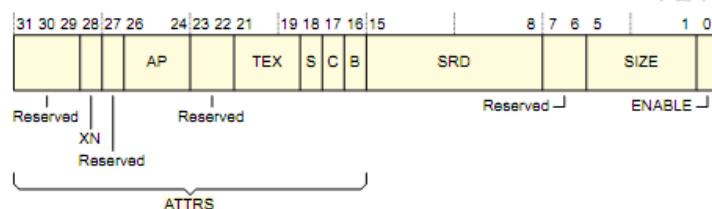
$$N = \text{Log}_2(\text{Ukuran region dalam byte})$$

Jika ukuran region ditentukan 4 GB, melalui register RASR, maka tidak bit-bit ADDR tidak akan valid. Dalam hal ini region akan menempati semua alamat memori Cortex-M3 dengan alamat awal berada di 0x00000000.

Bit VALID digunakan untuk memperbarui nomer region dan register RNR. Ketika bit ini bernilai 1, prosesor akan memperbarui register RNR dan memperbarui alamat awal dari region yang bersangkutan.

2.7.1.5 Register RASR

Register terakhir yang mengendalikan MPU adalah register RASR (*Region Attribute and Size Register*). Register ini digunakan untuk mengatur ukuran setiap region dan atribut setiap region memori dan juga mengaktifkan region dan sub-region. Region yang dipengaruhi adalah region yang ditunjukkan oleh register RNR.



Gambar 2.47 Register RASR

Bit XN (*Execute Never*), bit ke-28, digunakan untuk mengatur apakah region memori yang disebut oleh register RNR bisa melakukan eksekusi instruksi atau tidak. Bit ini di-set, maka region memori tersebut bisa melakukan eksekusi instruksi.

Bit-bit AP[2:0] (bit ke-24 – ke-26) merupakan bit-bit yang mengatur ijin akses (*Access Permission*) terhadap region memori yang ditunjukkan oleh register RNR. Bit-bit ini menentukan ijin akses terhadap sebuah region memori pada saat mode privileged dan mode user. Dengan bit-bit ini sebuah region memori bisa diatur untuk akses penuh (baca dan tulis), hanya baca atau tidak bisa diakses.

Bit-bit TEX[2:0] (bit ke-19 – ke-21) bersama-sama dengan bit S (*Shareable*), C (*Cacheable*), dan B (*Bufferable*) menentukan atribut dari sebuah region memori. Bit-bit ini akan mengatur memori cache (*cacheable*), pemakaian bersama memori oleh beberapa bus (*shareable*) dan juga penggunaan buffer saat akses memori. ARM Cortex-M0/M0+, M2, M3 dan M4 tidak

dirancang untuk mendukung memori cache, tetapi struktur register ini tetap mengacu ke arsitektur ARMv7-M. Jenis ARM Cortex-M7, yang baru dirilis oleh ARM Ltd, sudah mendukung memori cache.

Tabel 2.10 Ijin Akses Region Memori

AP[2:0]	Akses Privileged	Akses user	Keterangan
000	Tidak ada akses	Tidak ada akses	Mencoba akses ke region memori akan membangkitkan fault (MemManage)
001	Baca dan tulis	Tidak ada akses	Hanya bisa diakses di mode privileged
010	Baca dan tulis	Hanya baca	Operasi tulis di mode user akan membangkitkan fault (MemManage)
011	Baca dan tulis	Baca dan tulis	Akses penuh
100	Tidak bisa diprediksi	Tidak bisa diprediksi	Tidak dipakai
101	Hanya baca	Tidak ada akses	Region memori hanya bisa dibaca di mode privileged
110	Hanya baca	Hanya baca	Region memori hanya bisa dibaca
111	Hanya baca	Hanya baca	Region memori hanya bisa dibaca

Di arsitektur ARMv6 dan v7, sistem memori bisa mempunyai 2 level cache: cache dalam (*inner cache*) dan cache luar (*outer cache*) yang mempunyai pengaturan cache yang berbeda. Oleh karena ARM Cortex-M3 tidak mempunyai kendali cache, pengaturan cache hanya berefek kepada proses buffer di Matriks bus internal dan mungkin di pengendali memori.

Bit-bit SRD[7:0] (*Sub-Region Disable*) menentukan apakah region memori mendukung sub-region atau tidak. Bit SRD[0] mengatur region 0, SRD[1] untuk region 1 dan seterusnya. Jika bit SRD bernilai 0, maka region yang bersangkutan mendukung sub-region, dan jika SRD di-set 1, region tidak mendukung sub-region. Region dengan ukuran 128 byte tidak mendukung sub-region, oleh karena itu untuk region 32 byte, 64 byte dan 128 byte bit SRD harus bernilai 1.

Bit-bit SIZE[5:0] merupakan bit-bit untuk mengatur ukuran setiap region memori, nomor region yang diatur ditunjukkan oleh register RNR. Ukuran region memori dirumuskan sebagai:

Ukuran Region Memori = $2^{(\text{SIZE}+1)}$

Ukuran region terkecil yang diizinkan adalah 32 byte (SIZE bernilai 4) dan ukuran region terbesar yang memungkinkan 32 GB (SIZE bernilai 30).

Bit ENABLE, sesuai dengan namanya, adalah bit yang akan mengaktifkan region memori.

2.7.2 PENGGUNAAN MPU

Berdasarkan pemetaan memori (gambar 2.13), ARM Cortex-M3 mempunyai beberapa memori sebagai berikut:

1. Flash/ROM
2. SRAM Internal
3. SRAM Eksternal
4. Periperal
5. Periperal privat

Keempat memori di atas bisa mempunyai atribut memori sebagai berikut:

1. *Strongly-Ordered*
2. Device
3. Normal
4. Dapat dipakai bersama (*shareable*)

Atribut memori *strongly-ordered* digunakan untuk region periperal privat, seperti NVIC, timer Systick dan SCB (*System Control Block*). Pada dasarnya tidak diperlukan pengaturan MPU di blok memori ini. MPU akan secara otomatis mengenali alamat yang termasuk ke area periperal privat dan akan mengijinkan akses ke area ini di mode privileged. Atribut *strongly-ordered* tidak mengenal penulisan melalui buffer. Untuk memori yang lain bisa di-set berdasarkan tabel di bawah berdasarkan register RASR.

Ketika akan mengatur MPU, jika sebelumnya MPU sudah diprogram, region-region yang tidak dipakai harus di-non aktifkan terlebih dahulu agar setting MPU yang sudah ada tersebut tidak terpengaruh oleh setting MPU yang baru. Selain itu, semua interupsi harus dimatikan ketika memperbarui atribut memori, untuk menjaga jangan sampai ada ISR yang mengakses MPU saat proses perubahan terjadi.

Tabel 2.11 Seting Atribut Memori

Memori	TEX	C	B	S	Keterangan
--------	-----	---	---	---	------------

Flash/ROM	0000	1	0	0	Memori Normal, tidak bisa dipakai bersama, penulisan langsung
SRAM Internal	0000	1	0	1	Memori normal, dipakai bersama, penulisan langsung
SRAM Eksternal	0000	1	1	1	Memori normal, dipakai bersama, penulisan alokasi
Periperal	0000	0	1	1	Memori device, dipakai bersama

2.8 FITUR LAIN CORTEX-M3

ARM Cortex-M3 juga dilengkapi dengan fitur-fitur:

1. Timer Systick
2. Manajemen Daya
3. Komunikasi Multiprosesor
4. Kendali Reset

2.8.1 TIMER SYSTICK

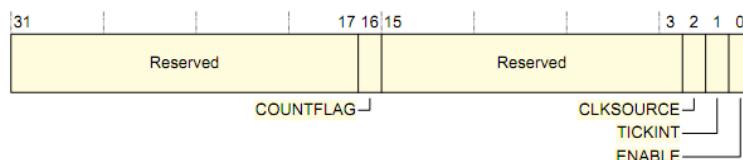
Seperti telah disinggung di sub-bab sebelumnya, Timer Systick merupakan timer 24 bit yang menjadi bagian dari core Cortex-M3. Timer ini merupakan timer yang mencacah mundur (*down counter*), artinya ketika nilai cacahan mencapai nol, akan kembali mencacah dari nilai awal yang diambil dari register reload.

Timer Systick biasa digunakan sebagai timer yang menyediakan fungsi pewaktuan pada sistem yang menggunakan RTOS, yang akan membangkitkan eksepsi secara periodik agar kernel OS melakukan konteks switch. Selain itu bisa juga digunakan di fungsi yang menyediakan fungsi tunda (delay).

Timer Systick bisa mengambil sumber clock internal (sama dengan clock yang dipakai oleh core) atau clock eksternal. Mungkin sebagian besar prosesor ARM Cortex-M3 tidak menyediakan clock eksternal untuk Timer Systick, karena biasanya eksternal clock digunakan juga memberikan clock untuk core. Sehingga perlu dilihat dari datasheet prosesor yang dikeluarkan oleh pabrik silikon yang membuatnya. Karena clock berasal dari clock prosesor (core), ada kemungkinan clock tersebut berhenti bekerja saat prosesor berada di kondisi mode daya rendah, sehingga timer Systick juga akan berhenti mencacah.

Timer Systick dikendalikan oleh 4 buah register

1. Register kendali dan status (SYST_CSR) untuk mengatur sumber clock, mengaktifkan pencacah, mengatur interupsi dan memberikan keadaan/ status dari timer Systick
2. Register nilai *reload* (SYST_SVR), register yang berisi nilai auto reload yaitu nilai cacahan awal setiap kali timer mencacah sampai nol.
3. Register yang berisi nilai cacahan yang sedang berjalan (SYST_CVR)
4. Register kalibrasi (SYST_CALIB), yang menunjukkan nilai awal untuk clock pewaktuan sebesar 10 mdetik (100Hz).



Gambar 2.48 Register CSR

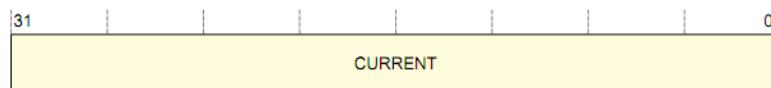
Register CSR (*Control and Status Register*) berisi bit-bit untuk mengendalikan timer Systick. Bit ENABLE digunakan untuk menghidupkan atau mematikan fungsi pencacah timer Systick. Men-set bit ENABLE akan menghidupkan fungsi timer.

Bit TICKINT menunjukan ketika cacahan mencapai 0, timer Systick akan mengubah status eksepsi timer Systick menjadi tertunda (pending) atau tidak. Jika bit ini 0, maka ketika cacahan mencapai 0, tidak akan mempengaruhi status eksepsi timer Systick (artinya tidak akan membangkitkan eksepsi), sedangkan jika di-set menjadi 1, bit ini akan mempengaruhi eksepsi timer Systick menjadi pending ketika cacahan mencapai 0 dan akan membangkitkan eksepsi/interupsi timer Systick.

Bit CLKSOURCE digunakan untuk memilih sumber clock timer Systick diambil dari clock eksternal (bila bit bernilai 0) atau diambil dari clock prosesor (CLKSOURCE=1). Untuk bit ini perlu melihat ke lembaran data dari pembuat prosesor untuk implementasi sesungguhnya. Sebagai contoh untuk prosesor STM32F207 dari ST Microelectronics, jika CLKSOURCE=1, clock diambil dari clock prosesor (AHB) sedangkan jika CRLSOURCE=0, clock berasal dari AHB/8. Sedangkan bit COUNTFLAG akan di-set menjadi 1 jika cacahan mencapai 0. Pembacaan terhadap bit COUNTFLAG akan membuat bit bernilai 0.

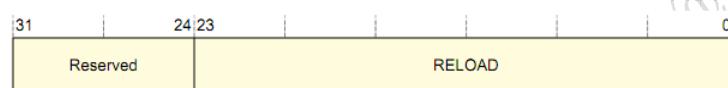
Ketika timer Systick pertama kali diaktifkan, bit ENABLE di-set, timer akan mulai mencacah mundur dengan nilai awal diambil dari register CVR (*Current Value Register*). Setelah mencapai 0, timer akan mencacah

mundur lagi dengan nilai awal diambil dari register RVR (*Reload Value Register*). Begitu seterusnya. Setiap kali cacahan mencapai 0, bit COUNTFLAG akan di-set.



Gambar 2.49 Register CVR

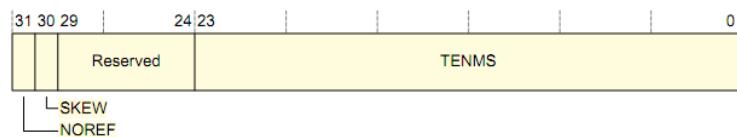
Operasi penulisan terhadap register CVR akan membuat nilai CURRENT menjadi 0 dan juga mengubah bit COUNTFLAG menjadi 0. Selain itu, akan menyebabkan timer melakukan reload (mencacah dari awal) dengan nilai awal diambil dari register RVR. Penulisan terhadap register CVR tidak akan memicu eksepsi timer Systick. Pembacaan terhadap register RVR akan menghasilkan nilai cacahan saat pembacaan dilakukan.



Gambar 2.50 Register RVR

Operasi penulisan dengan nilai 0 terhadap register RVR akan membuat timer Systick berhenti bekerja. Nilai register RVR tidak bisa ditentukan setelah reset, sebelum mengaktifkan timer Systick, software harus mengisi register RVR dengan nilai yang diinginkan, dan menulis ke register CVR. Hal ini akan membuat register CVR menjadi 0, saat diaktifkan, karena sudah 0, timer langsung melakukan reload register CVR dengan nilai dari register RVR dan mulai mencacah mundur.

Register CALIB (*Calibration*) digunakan untuk mengkalibrasi timer Systick untuk mendapatkan sistem pewaktuan yang presisi, misalnya 1 milidetik. Register CALIB merupakan register hanya baca. Bit NOREF menunjukkan apakah timer Systick menggunakan clock referensi atau tidak. Bit SKEW digunakan untuk menunjukkan apakah nilai TENMS sudah tepat untuk cacahan 10 mili detik (100Hz) atau tidak. Jika bit SKEW di-set 1, maka nilai TENMS bisa dijadikan sebagai nilai untuk register RVR untuk pewaktuan 10 mili detik.



Gambar 2.51 Register CALIB

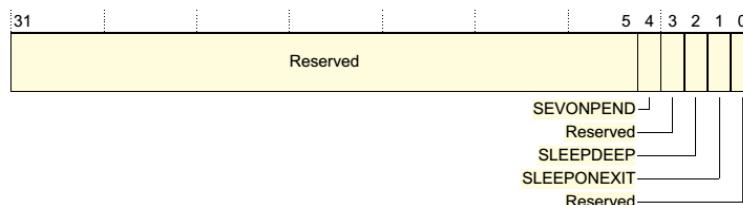
2.8.2 MANAJEMEN DAYA

Prosesor ARM adalah prosesor yang sejak awal dirancang untuk bekerja dengan konsumsi daya yang rendah. Pemilihan jenis prosesor dengan instruksi yang dikurangi (RISC) menjaga agar pabrikasi prosesor tidak membutuhkan rangkaian logika yang rumit dan jumlah gate yang rendah sehingga konsumsi daya tetap rendah. Selain itu juga menambahkan instruksi-instruksi untuk mode penghemat daya.

Prosesor ARM Cortex-M3 dilengkapi dengan fungsi atau mode 'tidur' (*sleep mode*). Pada saat berada di mode sleep, clock ke core Cortex-M3 berhenti bekerja tetapi ke periperal tetap, sehingga prosesor bisa dibangunkan melalui sebuah interupsi. walaupun demikian apa yang terjadi ketika mode sleep tergantung dari pabrik silikon yang membuat prosesor tersebut. Bisa hanya menghentikan clock ke prosesor, tetapi ke periperal tetap jalan, atau semua clock dimatikan. Kalau semua clock sudah dimatikan, hanya reset yang bisa membangunkan kembali prosesor dari mode tidurnya.

ARM menyediakan instruksi WFI (*Wait For Interrupt*) atau WFE (*Wait For Event*) untuk membuat prosesor masuk ke mode sleep. Ada 2 cara prosesor memasuki mode sleep setelah instruksi WFI dan WFE yang ditentukan oleh bit SLEEPONEXIT register SCR (*System Control Register*), yaitu:

1. Jika bit SLEEPONEXIT bernilai 0, maka prosesor akan langsung memasuki mode sleep segera setelah instruksi WFI atau WFE
 2. Jika bit SLEEPONEXIT di-set, prosesor akan memasuki mode sleep segera setelah keluar dari ISR dengan prioritas paling rendah.



Gambar 2.52 Register SCR

Untuk keluar dari mode sleep, ditentukan oleh instruksi apa yang menyebabkan prosesor memasuki mode sleep. Jika prosesor memasuki mode sleep karena instruksi WFI, maka semua interupsi periperal yang dikenali oleh kendali interupsi NVIC bisa membangunkan prosesor dari mode sleep. Prioritas interupsi tentunya harus lebih tinggi dari prioritas yang sekarang sedang terjadi atau yang di-set oleh register BASEPRI atau register mask (PRIMASK dan FAULTMASK).

Jika instruksi WFE yang membuat prosesor ke dalam mode sleep, maka untuk membangunkannya diperlukan sebuah *event* yang bisa berupa interupsi, interupsi yang terpicu sebelumnya atau event dari eksternal berupa pulsa ke sinyal RXEV (*Receive Event*). Sebuah interupsi masih bisa membangunkan prosesor walaupun prioritasnya lebih rendah daripada register BASEPRI atau register mask asalkan bit SEVONPEND di register SCR di-set.

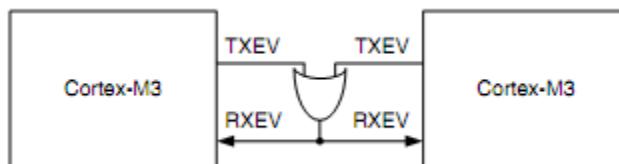
Selain dengan mengaktifkan mode sleep, penghematan daya pada prosesor ARM Cortex-M3, bisa dilakukan dengan cara:

1. Mengurangi frekuensi clock prosesor
2. Mematikan clock ke periperal-periperal yang tidak digunakan, pada umumnya setelah reset semua periperal dalam kondisi tidak aktif

2.8.3 KOMUNIKASI MULTIPROSESOR

Prosesor ARM Cortex-M3 dilengkapi dengan antarmuka komunikasi multiprosesor yang sedarhana untuk komunikasi event. Prosesor mempunyai 1 sinyal keluaran yang dinamakan *Transmit Event* (TXEV), untuk mengeluarkan sebuah event, dan sebuah masukan event, yang dinamakan dengan (RXEV) untuk menerima sinyal event. Hubungan komunikasi event antar 2 prosesor Cortex-M3 ditunjukkan oleh gambar 2.53.

Komunikasi event ini bisa digunakan ketika kedua prosesor masuk ke mode sleep melalui instruksi WFE. Ketika sebuah event terjadi maka salah satu prosesor bisa kembali ke mode normal dan dengan instruksi SEV (*Send Event*) bisa membangunkan prosesor yang lain secara bersamaan.



Gambar 2.53 Komunikasi Multiprosesor

2.8.4 KENDALI RESET

ARM Cortex-M3 mempunyai 2 kendali reset, yaitu melalui bit SYSRESETREQ (*System Reset Request*) di register NVIC dan melalui bit VECTREST di register AICR (*Application Interrupt and Reset Control Register*). Bit SYSRESETREQ memungkinkan prosesor ARM Cortex-M3 mengirimkan sinyal reset ke pembangkit reset. Pembangkit reset ini bukan bagian dari prosesor Cortex-M3, sehingga implementasinya tergantung kepada manufaktur silikon.

Sedangkan bit VECTREST, jika bit ini di-set ke 1 akan me-reset prosesor Cortex-M3, kecuali sistem debug. Sinyal reset VECTREST juga tidak akan me-reset periperal yang bukan bagian dari prosesor Cortex-M3. Misalnya jika prosesor tersebut mempunyai sebuah port serial (UART) maka VECTREST tidak akan me-reset UART tersebut. VECTREST ini ditujukan untuk melakukan debug saat mengembangkan program, di mana hanya diperlukan me-reset prosesor tetapi tidak sampai mereset sistem.

SYSRESETREQ sebaiknya digunakan ketika melakukan reset secara software untuk meyakinkan bahwa semua sistem akan mereset secara bersamaan. Untuk keterangan terperinci mengenai SYSRESETREQ, apakah juga akan mereset periperal, sebaiknya mengacu kepada lembaran data yang dikeluarkan oleh pembuat prosesor yang bersangkutan.

Ketika SYSRESETREQ dibangkitkan, prosesor mungkin tidak langsung melakukan reset, sehingga akan ada waktu tunda. Kemungkinan pada saat itu prosesor masih menerima interupsi setelah sinyal reset dikeluarkan. Untuk itu, agar prosesor tidak lagi menerima sinyal interupsi bisa dilakukan dengan mengatur interupsi melalui register PRIMASK atau FAULTMASK sebelum mengirimkan sinyal SYSRESETREQ.

2.9 SISTEM DEBUG

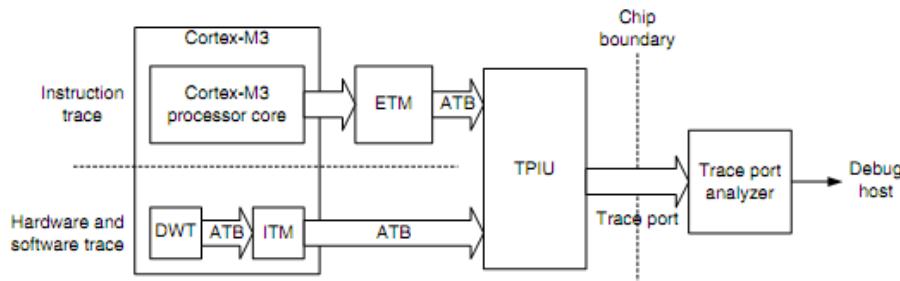
Debug merupakan proses penting dalam pengembangan software baik software yang berjalan di sebuah komputer (PC) maupun software yang dibuat untuk sistem embedded. Debug merupakan proses pencarian kesalahan (*bug*) yang mungkin terjadi ketika software sedang berjalan. Kata bug dipopulerkan di dunia komputer oleh seorang wanita programmer dan juga perwira angkatan laut Amerika bernama Grace Hopper. Pada saat itu tahun 1947, Grace sedang bekerja dengan komputer Mark II, sebuah komputer elektro mekanikal menggunakan relay-relay. Grace menemukan ada seekor serangga (sejenis kupu-kupu) yang tersangkut di dalam sebuah relay Mark II sehingga mengganggu kerja sistem. Ketika itu Grace mengatakan bahwa serangga tersebut telah men-debugging sistem. Sejak saat itu bug dan debugging menjadi terminologi di dunia komputer (software), walaupun sebenarnya bug telah juga digunakan dalam dunia engineering untuk menyatakan kesalahan kecil atau masalah yang tidak diharapkan.

Sistem debug di ARM Cortex-M3 menggunakan arsitektur *CoreSight* dan dilengkapi dengan komponen-komponen debug untuk kendali debug, penelusuran (*trace*) program dan sebagainya. Sistem debug ARM Cortex-M3 mendukung fitur-fitur debug seperti:

1. Memberhentikan program
2. Menjalankan program langkah demi langkah (*stepping*)
3. Titik berhenti (*breakpoint*)
4. Melihat nilai-nilai variabel yang digunakan dalam memori (SRAM) termasuk alamat memori yang digunakan
5. Penelusuran instruksi (*instruction trace*) melalui sebuah modul yang dinamakan *Embedded Trace Module* (ETM)
6. Penelusuran data
7. Penelusuran software melalui *Instrumentation Trace Module* (ITM)
8. Profiling melalui *Data Watchpoint and Trace* (DWT)

Sistem debug ARM Cortex-M3 dikendalikan oleh sebuah antarmuka bus yang dinamakan dengan *Debug Access Port* (DAP), yang pada dasarnya adalah sama dengan bus periperal (APB). Fungsi debug sebenarnya dikendalikan oleh NVIC dan beberapa komponen debug lainnya seperti ETM, *Flash Patch and Breakpoint* (FPB), ETM, ITM dan *Trace Port Interface Unit* (TPIU). NVIC berisi beberapa register untuk proses debug terhadap

core seperti menghentikan program dan menjalankan program langkah demi langkah sedangkan komponen lain menyediakan fungsi-fungsi *watchpoint*, *breakpoint*, dan mengeluarkan pesan debug.



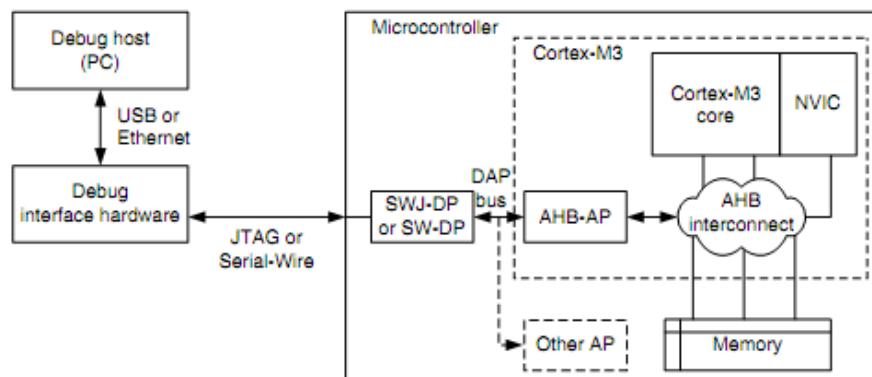
Gambar 2.54 Sitem Debug dan Penelusuran ARM Cortex-M3

Dengan adanya fungsi penelusuran (*trace*), ARM Cortex-M3 bisa di-debug secara real time artinya prosesor tetap dalam kondisi menjalankan programnya, fitur ini dinamakan *Debug Monitor Mode*. Berbeda dengan sistem debug biasa, di mana informasi mengenai debug hanya bisa dilakukan pada saat prosesor dihentikan dan program dijalankan langkah demi langkah, disebut juga *Halt Mode*. Mode debug monitor berguna pada beberapa aplikasi sistem embedded akan bermasalah pada saat debug kerja monitor dihentikan untuk memastikan bahwa sistem yang sedang di-debug tidak terjadi kerusakan hardware akibat kerja prosesor dihentikan.

Sistem telusur ARM Cortex-M3 terdiri atas:

1. Telusur instruksi melalui ETM
2. Telusur data melalui DWT
3. Pesan debug melalui ITM, yang kemudian meneruskan pesan debug ke TPIU.

Debug untuk pengembangan software PC bisa langsung dilakukan di software pengembangan tersebut (IDE), sedangkan debug di sistem embedded membutuhkan sebuah perangkat tambahan untuk menghubungkan IDE yang terinstal di PC dengan sistem embedded tersebut. Perangkat tambahan itulah yang kemudian mengakses perangkat-perangkat debug yang ada dalam prosesor dan menampilkannya ke program IDE di PC.



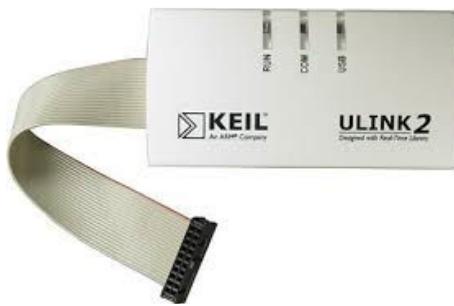
Gambar 2.55 Antarmuka Debug

ARM Cortex-M3, melalui DAP akan menghubungkan prosesor dengan perangkat debug eksternal melalui antarmuka JTAG (*Joint Test Action Group*) 5 pin atau melalui antarmuka 2 pin serial yang dinamakan dengan SWD (*Serial Wire Debug*). SWD dirancang karena ARM Cortex-M merupakan prosesor untuk aplikasi harga rendah sehingga ada yang dibuat dengan kemasan yang jumlah pin-nya sedikit. Sehingga port SWD lebih menghemat pin. Walaupun sebenarnya pin JTAG atau SWD bisa di-set sebagai pin IO biasa, tetapi sekali di-set fungsi pin debugnya tidak bisa digunakan lagi.

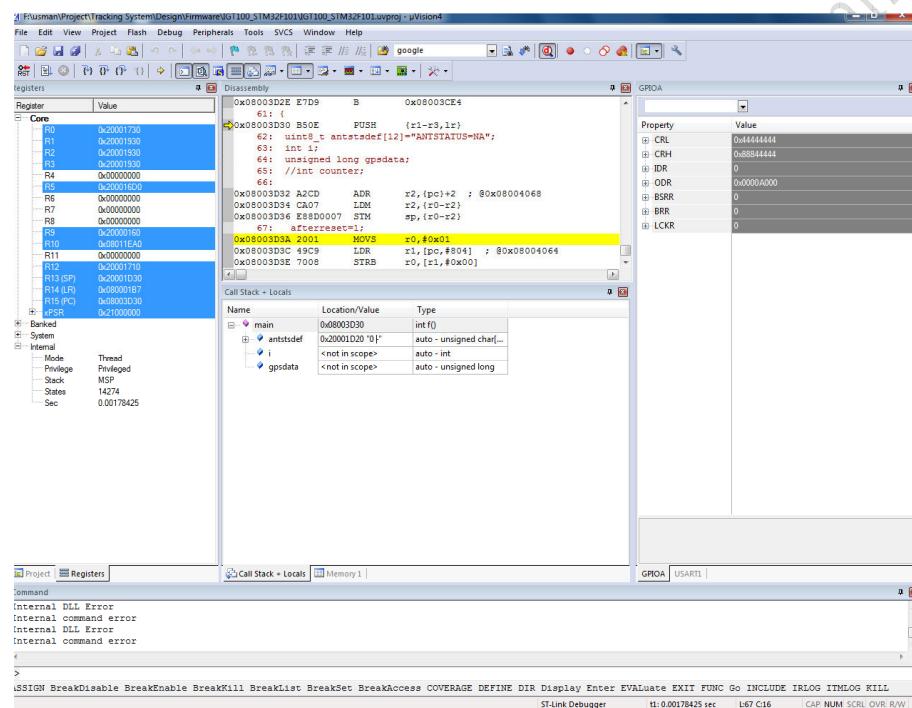
Perangkat debug biasanya terhubung dengan PC yang bertindak sebagai host melalui port USB yang ada di dalam PC. Sebelum USB berkembang, koneksi ke PC bisa melalui port serial, namun tentunya USB mempunyai kelebihan, salah satunya adalah transfer datanya yang lebih tinggi. Perangkat debug yang ada dipasaran untuk ARM Cortex-M biasanya bisa digunakan untuk koneksi melalui JTAG atau SWD. Salah satu contohnya adalah ST-Link dari ST Microelectronic, yang dirancang khusus untuk mikrokontroler ARM Cortex-M buatan ST Microelectronics. Atau ULINK2 dari Keil yang mendukung prosesor-prosesor ARM Cortex dari beberapa manufaktur. Perlu dicatat bahwa tidak semua perangkat debug bisa melakukan mode real time di mana prosesor tetap jalan, sebagai contoh ST-Link hanya mampu debug di mode halt, tetapi ST-Link V2 bisa melakukan debug di mode real time (trace).

IDE yang biasa dipakai untuk mengembangkan software prosesor ARM Cortex, seperti keil, IAR, CoIDE dan lain-lain, sudah dilengkapi dengan

fasilitas debugging dan mampu terhubung dengan berbagai macam perangkat debug seperti ST-Link atau ULINK2.



Gambar 2.56 ULINK Debugger dari Keil

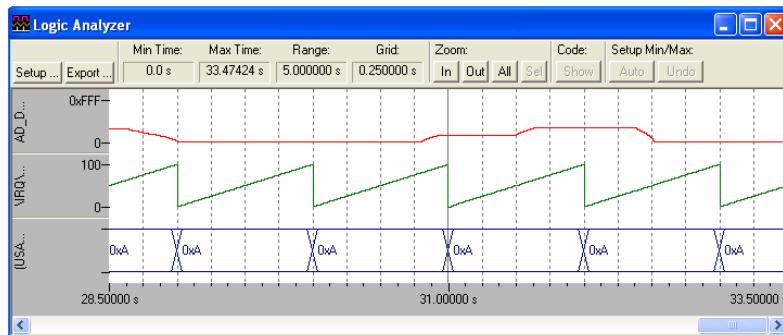


Gambar 2.57 IDE dari Keil di mode Debug

Selain untuk debug, port debug (JTAG dan SWD) bisa digunakan untuk mendownload program yang sudah dikembangkan melalui IDE dan

compiler C yang sesuai ke dalam memori flash. Hal ini bisa dilakukan melalui IDE langsung atau software tambahan. Karena untuk bisa melakukan debug, program harus didownload dulu ke dalam memori prosesor yang akan didebug.

Ketika proses debug berlangsung, maka IDE bisa melakukan instruksi langkah demi langkah, melihat isi register, periperal atau SRAM, seperti gambar 2.57. Ketika perangkat debug mendukung fungsi terlusur, maka IDE bisa menampilkan pewaktuan dari sebuah periperal internal misalnya ADC atau UART, seperti fungsi *Logic Analyzer* dari Keil.



Gambar 2.58 Logic Analyzer Menampilkan ADC dan UART

2.10 SET INSTRUKSI ARM DAN THUMB-2

Set-set instruksi merupakan bahasa assembly untuk ARM Cortex-M3. Walaupun buku ini semua contoh program menggunakan bahasa C, namun pada dasarnya bahasa C tersebut saat di-compile akan diubah menjadi bahasa assembly. Selain itu pada beberapa aplikasi yang membutuhkan kecepatan tinggi, masih menggunakan bahasa assembly atau digabung antara bahasa C dengan bahasa assembly (*in-line assembly*). Dan bab ini akan membahas secara singkat set-set instruksi yang didukung oleh prosesor ARM Cortex-M3.

Prosesor ARM sejak awal dirancang untuk bekerja dalam 32 bit, sehingga semua instruksinya menggunakan instruksi 32 bit. Kelebihan instruksi 32 bit adalah di kecepatannya. Misalnya dalam operasi aritmatika yang melibatkan bilangan 32 bit (4 byte), sebuah prosesor 8 bit (1 byte) akan membutuhkan 4 langkah untuk menyelesaikan operasi tersebut, tetapi prosesor 32 bit cukup membutuhkan 1 langkah untuk menyelesaikannya.

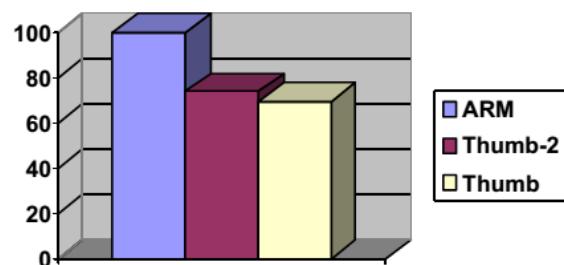
Kekurangan dari instruksi 32 bit adalah tentu saja, setiap instruksi akan membutuhkan ruangan 4 byte di memori program.

Pada tahun 1993, raksasa pembuat handphone pada saat itu Nokia, sekarang sudah diakuisisi Microsoft, meminta TI (*Texas Instruments*) untuk memproduksi chipset untuk produk handphone terbarunya. TI kemudian menawarkan chipset berbasis prosesor ARM7 kepada Nokia. Namun Nokia menolak mengingat ARM7 memerlukan 4 byte untuk setiap instruksinya dan tentu saja memerlukan memori program yang lebih besar sehingga membuat harga chipset menjadi lebih mahal. Hal ini kemudian membuat ARM mengembangkan instruksi 16 bit untuk prosesor ARM7. Dan instruksi 16 bit bisa menghemat pemakaian memori sampai 35%, sehingga ukuran memori bisa dikurangi menyamai mikrokontroler 16 bit. Nokia pun setuju dan lahirlah handphone pertama yang menggunakan prosesor ARM yaitu Nokia 6110.

Instruksi 16 bit itu dinamakan dengan instruksi *Thumb*, sementara instruksi 32 bit dinamakan dengan instruksi ARM. Selain hemat memori, penggunaan instruksi Thumb juga bisa menghemat pemakaian daya prosesor. Tapi tentu saja instruksi Thumb, demi penghematan memori, akan lebih lambat bila dibandingkan dengan instruksi ARM. Selain itu, dengan instruksi Thumb tidak semua register prosesor bisa diakses. R0 sampai R7 bisa diakses oleh instruksi Thumb, tetapi register R8 – R15 hanya bisa diakses oleh beberapa instruksi Thumb (misalnya instruksi MOVE dan ADD).

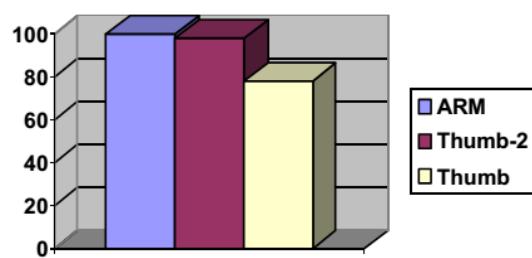
Dalam aplikasi, instruksi Thumb digunakan di fungsi-fungsi yang tidak memerlukan kecepatan yang tinggi. Sedangkan instruksi ARM digunakan untuk menangani fungsi-fungsi yang memerlukan kecepatan, misalnya saat menangani sebuah interupsi atau eksepsi. Untuk itu diperlukan pencampuran instruksi ARM dan instruksi Thumb yang prosesnya dinamakan dengan *ARM/Thumb Interworking*. Setelah reset prosesor akan berada di status ARM (*ARM state*), untuk mengubah status agar prosesor menjadi status Thumb (*Thumb State*) bisa dilakukan dengan beberapa cara. Pertama dengan cara manual menggunakan bahasa assembly atau ketika menggunakan compiler C maka *linker* ARM yang akan melakukan proses interworking melalui *Veneer* dengan menambahkan beberapa baris code ketika ada perubahan status prosesor baik dari ARM ke Thumb atau sebaliknya. Proses interworking ini akan mempengaruhi kinerja prosesor.

Untuk mengatasi kelemahan-kelemahan instruksi Thumb, ARM Ltd. kemudian menciptakan instruksi *Thumb-2*. Diperkenalkan pertama kali di core ARM1156 tahun 2003, instruksi *Thumb-2* merupakan perbaikan dari instruksi *Thumb*. Instruksi *Thumb-2* mengijinkan pencampuran antara instruksi ARM 32 bit dengan instruksi 16 bit tanpa adanya penambahan proses interworking, sehingga akan meningkatkan kinerja prosesor disamping menghemat pemakaian memori program.



Gambar 2.59 Ukuran Memori Instruksi Prosesor ARM

Instruksi *Thumb-2* mempunyai kinerja hampir 98% dari instruksi ARM atau 25% lebih baik dari instruksi *Thumb*. Dan kebutuhan memori program hanya 74% dari instruksi ARM 32 bit. Prosesor ARM Cortex-M3 pun sudah menggunakan instruksi *Thumb-2*.



Gambar 2.60 Perbandingan Kinerja Setiap Instruks

by Kang U2Man (u_2man@yahoo.co.id)

Bab 3

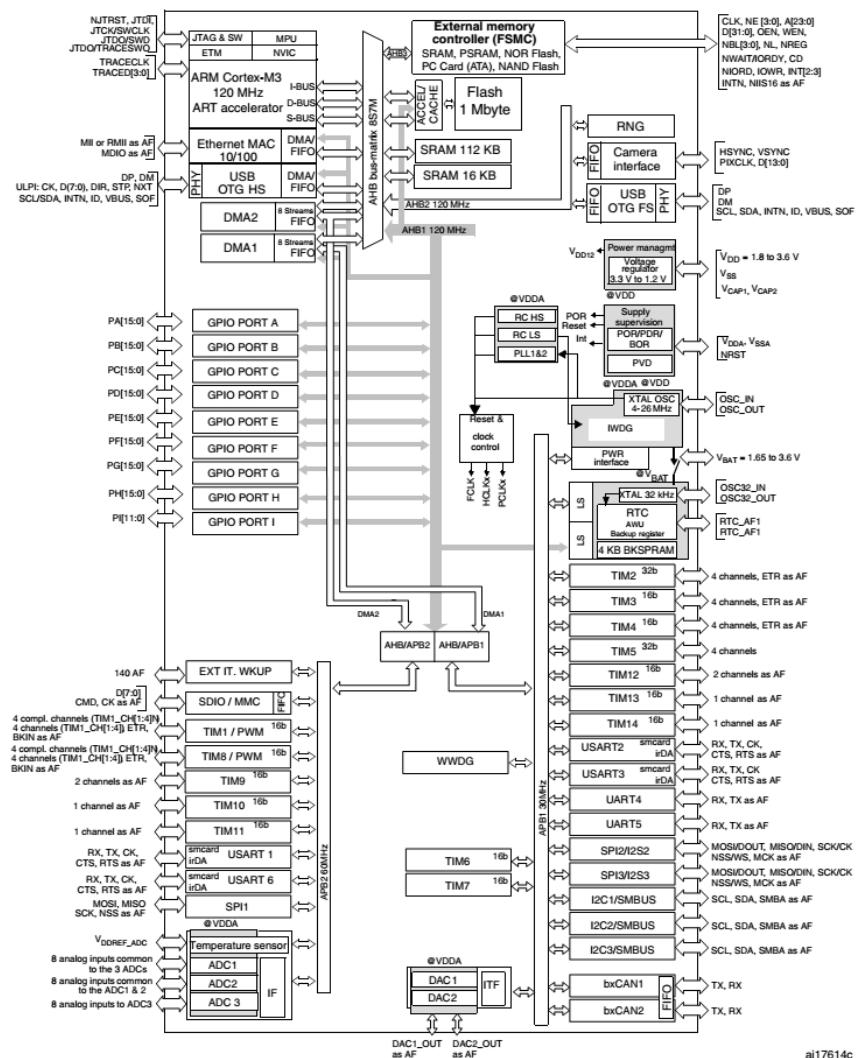
MIKROKONTROLER STM32F207

Mikrokontroler STM32F207 merupakan salah satu seri mikrokontroler dengan *core* ARM Cortex-M3 yang dikembangkan oleh ST Microelectronics. STM32F207 bisa bekerja sampai frekuensi maksimal 120 MHz lebih cepat bila dibandingkan dengan seri Cortex-M3 lainnya (STM32F1, STM32L1 atau STM32W). Dengan dilengkapi *Adaptive Real Time Accelerator* (ART Accelerator), STM32F207 mempunyai kecepatan 150 DMIPS (*Dhrystone Million Instructions Per Second*) atau 1.25 DMIPS/MHz. ART Accelerator, sesuai dengan namanya, ini akan mempercepat akses ke memori flash yang menyimpan program yang harus dijalankan. Proses pengambilan instruksi (*instruction prefetch*), pendekodean instruksi dan eksekusi instruksi bisa mencapai 0 *wait state* pada frekuensi 120 MHz.

STM32F207 bisa memiliki memori flash untuk menyimpan program dari 256 KB sampai 1 MB dan SRAM 128 KB. Selain itu, STM32F207 juga dilengkapi kendali memori FSMC (*Flexible Static Memory Controller*) untuk mengakses berbagai macam memori eksternal seperti *Compact Flash*, SRAM, PSRAM, NOR dan memori tipe NAND. FSMC juga bisa digunakan untuk mengakses LCD secara paralel dengan mode 8080 atau 6800.

STM32F207 diproduksi dengan berbagai macam periperal internal seperti ditunjukkan oleh diagram blok gambar 4.1. Beberapa periperal merupakan periperal yang biasa ada di sebuah mikrokontroler, seperti GPIO (*General Purpose Input/Output*), I2C (*Inter Integrated Circuit*), SPI (*Serial Peripheral Interface*), timer, ADC (*Analogue to Digital Converter*), DAC (*Digital to Analogue Converter*), port serial (UART) dan timer Watchdog. Periperal

lainnya untuk komunikasi yang cukup canggih seperti ethernet, untuk komunikasi dengan internet, USB yang mendukung *device* dan *host* serta OTG (*On The Go*). Ada juga periperal untuk mengakses kamera digital melalui port DCMI (*Digital Camera Interface*) yang bisa merekam video dan periperal untuk mengakses kartu memori (MMC, mikro SD atau Compact Flash) melalui port SDIO.



Gambar 3.1 Diagram Blok STM32F207

Selain itu ST Microelectronics melengkapi semua seri ARM Cortex-Mnya dengan kendali DMA (*Direct Memory Access*). Dengan adanya DMA ini, proses transfer data baik dari memori ke memori atau memori ke periperal bisa dilakukan tanpa pengawasan dari CPU. Artinya CPU bisa diprogram untuk mengerjakan tugas yang lain. Setelah proses transfer selesai, DMA tinggal diprogram untuk memberikan interupsi kepada CPU, sehingga CPU bisa memberikan respon yang diperlukan.

Periperal-periperal tersebut terkoneksi dengan 2 buah bus AHB (AHB1 dan AHB2) untuk periperal berkecepatan tinggi dan melalui 2 buah bus APB (APB1 dan APB2) untuk periperal berkecepatan rendah. Kedua APB ini kemudian terhubung dengan bus AHB melalui jembatan AHB ke APB. Sistem AHB terhubung ke core Cortex-M3 melalui sistem bus matrik yang juga akan menghubungkan dengan kendali DMA dan sistem memori. Kecepatan bus AHB (AHB1 dan AHB2) sama dengan kecepatan core (120 MHz) sedangkan kecepatan bus APB1 dibatasi di 30 MHz dan APB2 di 60 MHz.

Bab ini akan menjelaskan secara singkat tentang fitur-fitur serta periperal internal yang dimiliki oleh STM32F207. Untuk keterangan lebih detail mengenai fitur-fitur tersebut atau bagaimana cara mengakses berbagai periperal internal yang dimilikinya melalui register-register bisa dilihat dari datasheet dan *Reference Manual* yang tersedia di CD pendukung atau dapat diunduh dari website ST (www.st.com).

3.1 SISTEM ARSITEKTUR DAN MEMORI

3.1.1 SISTEM BUS MATRIX

Sistem arsitektur yang dimaksud di sini adalah bagaimana matrik bus AHB 32 bit menghubungkan *core* ARM Cortex-M3 dengan sistem memori dan berbagai periperal yang dimiliki oleh STM32F207. Sistem matrik ini merupakan matrik 8x7, 8 bus matrik master dan 7 bus *slave*. 8 matrik master terdiri atas:

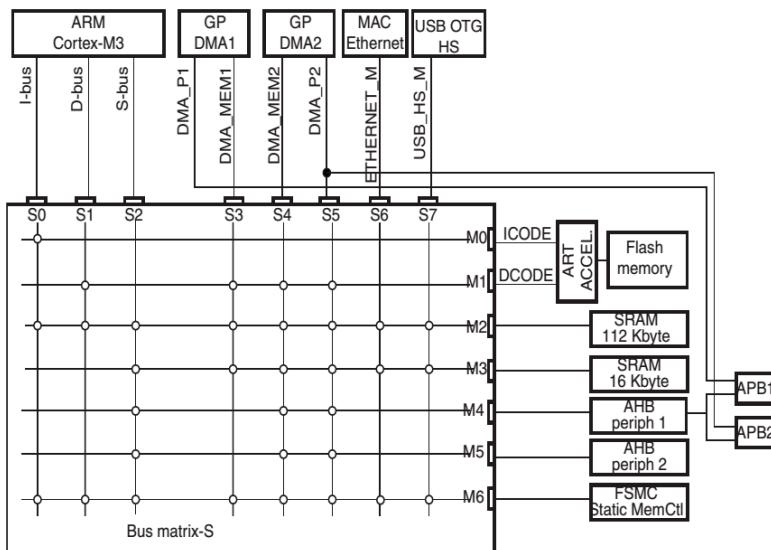
- 3 buah bus dari Cortex-M3, I-bus, D-bus dan S-bus (bus S0, S1, dan S2)
- Bus memori DMA1 (S3)
- Bus memori DMA2 (S4)
- Bus periperal DMA2 (S5)
- Bus DMA ethernet (S6)

- Bus DMA USB OTG HS (*High Speed*) (S7)

Sedangkan 7 bus slave terdiri atas:

- Bus ICode memori flash internal (M0)
- Bus DCode memori flash internal (M1)
- SRAM1 internal utama (112 KB) (M2)
- SRAM2 internal tambahan (16 KB) (M3)
- Periperal-periperal yang terhubung ke AHB1, termasuk jembatan AHB ke APB dan APB1 untuk periperal (M4)
- Periperal-periperal yang terhubung ke AHB2 (M5)
- FSMC (M6)

Sistem bus matrik ini ditunjukkan oleh gambar 3.2



Gambar 3.2 Sistem Bus Matrik STM32F207

Matrik S0 (*I-bus/Instruction bus*) menghubungkan core Cortex-M3 dengan sistem bus matrik sehingga CPU bisa mengambil instruksi program yang tersimpan di memori flash/SRAM internal atau pun memori eksternal yang terhubung melalui kendali FSMC. Matrik S1 (*D-bus/Data bus*) menghubungkan core Cortex-M3 dengan memori flash dan SRAM internal dan juga memori eksternal melalui FSMC. Bus ini bisa mengambil data atau instruksi. Proses debug akan menggunakan D-bus ini. Bus S2 (*S-bus/System bus*) ini akan menghubungkan Cortex-M3

dengan SRAM dan periperal STM32F207 yang terhubung melalui bus AHB dan APB. S-bus juga bisa digunakan untuk pengambilan instruksi walaupun kurang efisien bila dibandingkan dengan I-bus.

S3 dan S4 menghubungkan kendali DMA dengan memori SRAM internal atau eksternal (FSMC). S3 ini digunakan saat proses transfer data melalui DMA antar memori ke memori. S5 digunakan untuk mengakses periperal melalui DAM dan juga transfer data memori ke memori (internal maupun eksternal).

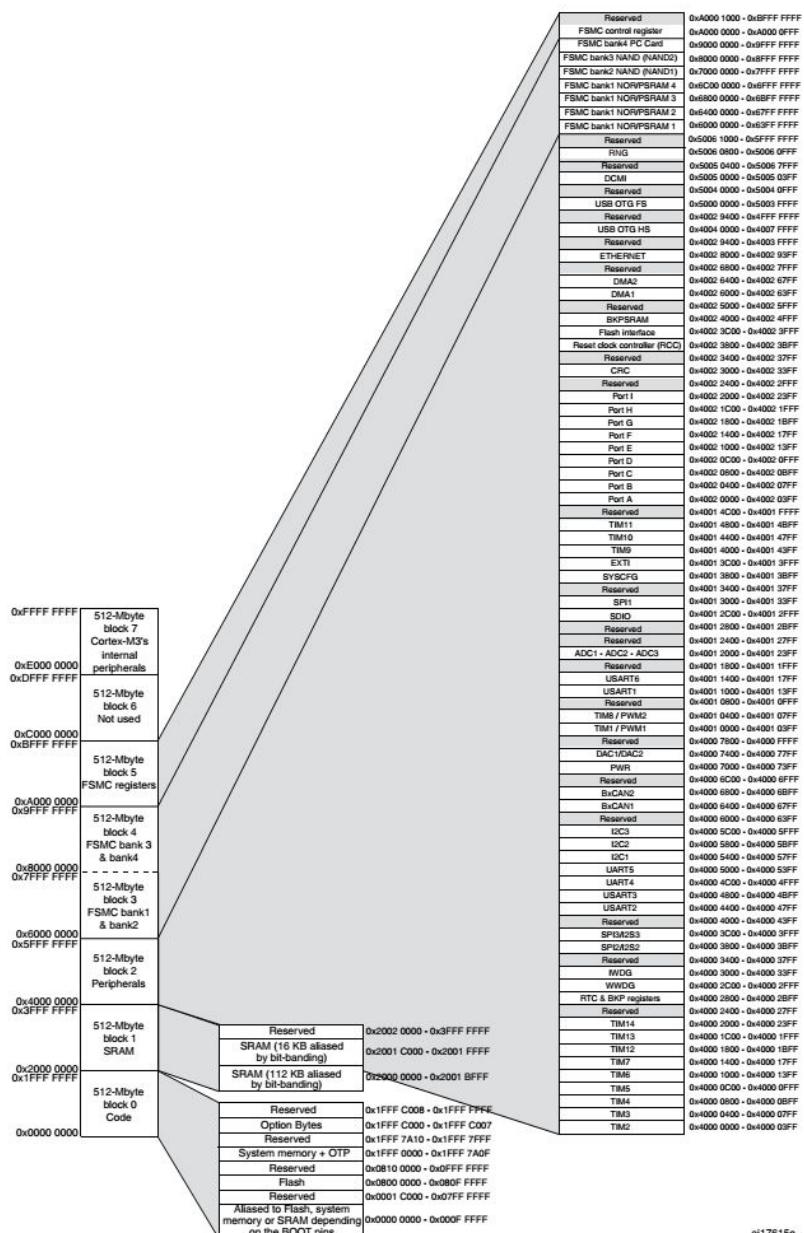
Bus S6 digunakan oleh kendali DMA ethernet untuk proses transfer data dari ethernet ke memori (internal atau eksternal). Bus S7 digunakan untuk koneksi DMA USB OTG HS ke memori (internal maupun eksternal).

3.1.2 PEMETAAN MEMORI

Memori STM32F207 mengikuti standar memori Cortex-M3 yang bisa mengalami sampai 4 Gbyte. Memori 4 Gbyte dibagi menjadi 8 blok yang masing-masing berukuran 0.5 Gbyte (512 Mbyte). Memori program (flash), internal SRAM, periperal internal atau memori eksternal akan berada di alamat 4 Gbyte ini.

Blok pertama (blok 0) merupakan blok code, di blok ini memori program (flash), *boot loader* (*System* memori) dan byte-byte pilihan (*option bytes*) ditempatkan. Seri STM32F207 bisa memiliki flash memori dari 256 Kbyte sampai 1 Mbyte. Boot loader merupakan sebuah program yang ditanam saat pabrikasi yang digunakan untuk mengakses area flash memori dan area byte-byte pilihan (memprogram,membaca atau menghapus). Sesaat reset, STM32F207 akan melakukan *boot* melalui blok 0 ini (alamat 0x00000000 sebagai vektor reset) tergantung dari konfigurasi pin Boot 0 dan Boot 1 seperti yang akan dibahas kemudian.

Blok kedua (blok1) merupakan blok area memori SRAM (internal). Seri STM32F207 memiliki 128 Kbyte. SRAM dibagi menjadi 2 blok yang masing-masing berukuran 112 Kbyte dan 16 Kbyte. SRAM ini tentunya mendukung operasi bit band dan akses tak rata seperti telah dibahas di bab sebelumnya. Selain itu STM32F207 memiliki tambahan SRAM sebesar 4 Kbyte yang dinamakan dengan SRAM cadangan (*Backup SRAM*). SRAM ini terhubung dengan pin VBAT (tegangan batere), sehingga walaupun catu daya utama mati, tetapi tegangan VBAT masih ada, data di SRAM ini tidak akan hilang.



ai17615c

Gambar 3.3 Pemetaan Memori STM32F207

Blok berikutnya ditempati oleh internal periperal dan kendali akses memori eksternal (FSMC). Seperti diketahui, dalam mikrokontroler

mengakses periperal artinya mengakses register yang mengatur periperal tersebut, biasanya sebuah periperal akan mempunyai register untuk mengatur (*setting register*), register yang menampilkan status dari periperal tersebut (*status register*) dan register yang merupakan register untuk mengirim atau membaca data ke periperal (*data register*).

Blok 3 sampai blok 5 merupakan area memori untuk akses ke memori eksternal melalui FSMC. Blok 5 merupakan area untuk register yang mengatur kerja FSMC, sedangkan blok 3 dan 4 merupakan area untuk bank memori FSMC. STM32F207 mempunyai 4 bank memori yang bisa dipakai untuk mengakses memori eksternal NAND, NOR atau PSRAM.

Blok terakhir tidak digunakan oleh STM32F207.

3.1.3 MEMORI FLASH

Seri STM32F207 dilengkapi flash memori untuk menyimpan program dari 256 Kbyte sampai 1 Mbyte. Flash memori ini menempati blok 0 dari sistem memori Cortex-M3 dan menempati alamat awal 0x80000000. Selain memori flash, blok 0 dibagi sebagai berikut:

- Memori flash itu sendiri, yang dibagi menjadi 12 sektor, masing-masing 4 sektor berukuran 16 Kbyte, 1 sektor berukuran 64 Kbyte dan 7 sektor berukuran 128 Kbyte. Total sampai 1 Mbyte.
- Memori sistem (*System Memory*), berukuran 30 Kbyte, merupakan ROM internal tempat menyimpan bootloader. Bootloader ini diprogramkan pada saat STM32F207 diproduksi. Fungsinya untuk mendownload program ke memori flash utama melalui jalur komunikasi UART, CAN, USB, I2C dan lain-lain.
- Memori OTP (*One-time programmable*), berukuran 528 byte, berguna untuk menyimpan data. Sepertinya tersirat dari namanya, memori OTP ini hanya bisa diprogram 1 kali. Untuk menyimpan data terdiri dari 512 byte, 16 byte sisanya digunakan untuk mengunci data tersebut.
- Byte-byte pilihan (*Option bytes*), berukuran 16 byte, merupakan memori untuk memproteksi memori flash (proteksi baca dan tulis), dan juga untuk mengatur fungsi reset *Brownout/BOR* (*Brownout reset*), timer *watchdog* dan fungsi reset saat MCU berada di mode *standby* atau stop.

Tabel 3.1 Pembagian Blok Memori Flash

Blok	Nama	Alamat Awal Blok	Ukuran
Memori Utama	Sektor 0	0x0800 0000 - 0x0800 3FFF	16 Kbyte
	Sektor1	0x0800 4000 - 0x0800 7FFF	16 Kbyte
	Sektor 2	0x0800 8000 - 0x0800 BFFF	16 Kbyte
	Sektor 3	0x0800 C000 - 0x0800 FFFF	16 Kbyte
	Sektor 4	0x0801 0000 - 0x0801 FFFF	64 Kbyte
	Sektor 5	0x0802 0000 - 0x0803 FFFF	128 Kbyte
	Sektor 6	0x0804 0000 - 0x0805 FFFF	128 Kbyte
	.	.	.
	.	.	.
	.	.	.
	Sektor 11	0x080E 0000 - 0x080F FFFF	128 Kbyte
	Memori Sistem	0x1FFF 0000 - 0x1FFF 77FF	30 Kbyte
OTP	OTP	0x1FFF 7800 - 0x1FFF 7A0F	528 byte
	Byte-byte Pilihan	0x1FFF C000 - 0x1FFF C00F	16 byte

Mikrokontroler STM32F207 menjalankan program dengan membaca program yang tersimpan di memori flash. Supaya proses pembacaan memori flash oleh core Cortex-M3 selalu benar, maka perlu diatur *latency* atau waktu tunggu (*wait states*) yang disesuaikan dengan frekuensi kerja dan tegangan dari mikrokontroler STM32F207. Pada dasarnya semakin tinggi frekuensi kerja, latency harusnya semakin panjang. Hal ini dilakukan dengan mengatur register FLASH_ACR (*Flash Access Control Register*).

Walaupun demikian, dengan adanya teknologi *Adaptive Real Time memory Accelerator* (ART Accelerator), yang dirancang untuk mikrokontroler seri STM32, maka waktu latency pembacaan flash untuk semua seri STM32 adalah 0 (tanpa latency) pada frekuensi clock maksimum, untuk STM32F207 adalah 120 MHz. ART Accelerator pada dasarnya akan mempercepat akses memori flash sehingga bisa mengimbangi kecepatan core Cortex-M3 pada frekuensi maksimumnya.

3.1.4 KONFIGURASI BOOT

Prosesor Cortex-M3 akan selalu mempunyai alamat vektor reset di area ICode (blok 0 di peta memori), yang beralamat awal di 0x0000 0000. Itulah kenapa memori flash ditempatkan di area kode (blok 0). Walaupun

memori flash tidak beralamat di 0x0000 0000 melainkan di alamat 0x0800 0000. Di peta memori alamat 0x0000 0000 (sampai ukuran 1 Mbyte, 0x000F FFFF) ditempati alamat *alias* tergantung dari pin BOOT dari STM32F207. Alamat alias ini bisa dipetakan (*remapping*) ke alamat flash, bootloader ataupun SRAM bahkan bisa juga ke memori eksternal.

STM32F207 mempunya dua pin BOOT (BOOT0 dan BOOT1). Pin BOOT0 merupakan pin khusus untuk konfigurasi boot ini, sedangkan BOOT1 bisa digunakan juga sebagai pin I/O biasa (GPIO/PB2). Kondisi pin BOOT ini dibaca saat clock ke-4 setelah reset. Tergantung kondisi keduanya, STM32 bisa melakukan booting melalui memori flash, bootloader, SRAM atau memori eksternal (FSMC), seperti tabel di bawah ini.

Tabel 3.2 Mode Booting

Kondisi Pin BOOT		Mode Boot	Alias
BOOT1	BOOT0		
x	0	Memori Flash	STM32F207 akan <i>boot</i> dari memori flash (menjalankan program dari memori flash)
0	1	Memori sistem (bootloader)	STM32F207 akan menjalankan program bootloader
1	1	SRAM	STM32F207 akan menjalankan program dari SRAM

Pin BOOT juga akan dibaca pada saat prosesor keluar dari mode *standby*, sehingga kedua pin BOOT ini sebaiknya berada pada kondisi logika yang sebenarnya. Dalam kebanyakan aplikasi, STM32F207 akan selalu menjalankan program dari memori flash, sehingga pin BOOT0 harus berada di logika 0 (rendah), hal ini bisa dilakukan dengan menghubungkan pin BOOT0 ke *ground* catu daya baik secara langsung atau melalui resistor *pull down*. Sedangkan pin BOOT1 bisa dipakai sebagai pin I/O biasa.

Ketika boot dipetakan ke memori sistem, maka STM32F207 akan menjalankan program bootloader yang telah diprogramkan pada saat pabrikasi STM32F207. Tugas utama bootloader ini adalah untuk program ke memori flash melalui periperal komunikasi yang dimiliki oleh STM32F207 yaitu USART, CAN atau USB, tergantung dari versi bootloadernya. Sehingga tidak diperlukan hardware tambahan saat mendownload program ke memori flash.

3.2 GPIO

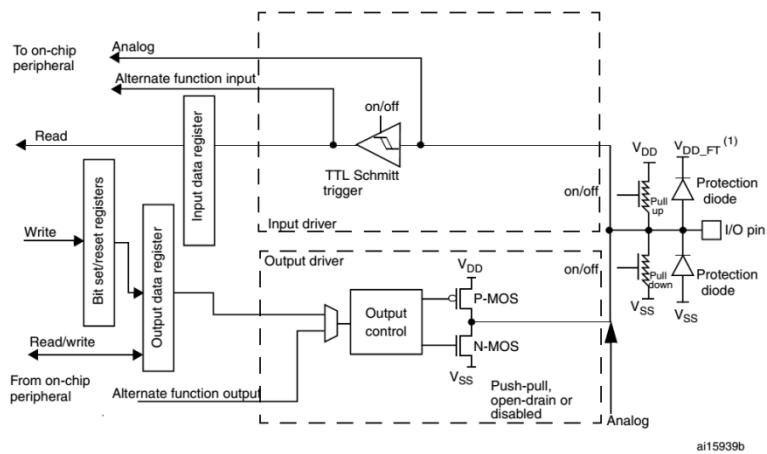
STM32F207 diproduksi pada kemasan LQFP (*Low Profile Quad Flat Package*) 64, 100,176 dan 144 pin, kemasan UFBGA (*Ultra thin Fine Pitch Ball Grid Array*) 176 pin dan kemasan WLCSP (*Wafer Level Chip Size Package*) 64 pin. Di kemasan 176 pin (LQFP176 atau BGA176), STM32F207 bisa mempunyai sampai 140 pin GPIO (*General Purpose Input/Output*). GPIO ini dinamai dengan PA (Port A), PB (Port B), PC (Port C), PD (Port D), PE (Port E), PF (Port F), PG (Port G), PH (Port H) dan PI (Port I).

3.2.1 FITUR-FITUR

Setiap port bisa mempunyai sampai 16 pin I/O (tergantung kemasan). Setiap pin bisa dikonfigurasi secara individu sebagai input atau output dengan mode *open drain* atau *push pull*, sebagai pin fungsi alternatif ($AF = Alternative Function$), dengan atau tanpa *pull-up* atau *pull-down* atau sebagai sumber interupsi eksternal (EXTI). Ketika difungsikan sebagai input/output bisa dikendalikan sebagai port (16 bit) atau sebagai pin individual. GPIO ini terhubung langsung dengan bus AHB1 yang mempunyai frekuensi sampai 120 MHz, sehingga GPIO bisa mempunyai kecepatan *toggle* sampai 60 MHz.

GPIO STM32F207 mempunyai fitur *5V tolerant* artinya walaupun STM32F207 dicatuh daya dengan tegangan 3.3V, tetapi setiap port bisa dengan aman dihubungkan dengan sistem logika TTL yang bertegangan 5V. Tapi hal ini tidak berlaku jika port atau pin difungsikan sebagai pin ADC atau DAC di mana tegangan maksimumnya hanya 3.3V, tegangan analog (VDDA) dan tegangan referensinya (VREF).

Pin-pin GPIO, seperti halnya dengan mikrokontroler lain, bisa difungsikan untuk fungsi alternatif. Fungsi alternatif ini akan menjadikan sebuah pin untuk menjadi masukan atau keluaran dari periperal internal, misalnya pin Rx (penerima) atau Tx (pengirim) dari sebuah port serial (UART). Konfigurasi pin untuk fungsi alternatif ini telah ditetapkan pada saat produksi STM32F207, misalnya sebuah pin hanya bisa dijadikan sebagai pin untuk UART, Timer atau fungsi USB, sementara untuk fungsi ADC harus menggunakan pin yang lainnya.



Gambar 3.4 Struktur Dasar GPIO

Semua port (atau pin) juga bisa digunakan sebagai sumber interupsi eksternal (EXTI) ketika pin tersebut difungsikan sebagai input. Interupsi bisa diatur untuk transisi logika rendah ke logika tinggi (*rising edge*) atau transisi dari logika tinggi ke logika rendah (*falling edge*) atau kondisi keduanya, membangkitkan interupsi saat transisi dari logika rendah ke logika tinggi dan saat transisi dari logika tinggi ke logika rendah.

Semua konfigurasi GPIO, yang tersimpan di dalam register-register pengaturan GPIO, bisa dikunci (*lock*) untuk menjaga agar konfigurasi ini tidak bisa diubah secara sembarangan. Fungsi penguncian ini dilakukan dengan melakukan penulisan dengan urutan tertentu ke register pengunci (register GPIOx_LCKR).

3.2.2 PENGATURAN GPIO

Penggunaan GPIO diatur oleh 10 buah register, 7 buah register untuk fungsi pengaturan, 2 buah register untuk input dan output data dan 1 buah register untuk untuk men-set atau me-reset pin I/O secara bit individual. Setiap port (PA sampai PI) mempunyai susunan regiser masing-masing, sehingga total ada 90 buah register yang mengatur GPIO. Semua register merupakan register 32 bit, namun bisa diakses sebagai byte (8 bit), *half-word* (16 bit) maupun *word* (32 bit).

3.2.2.1 Register Mode GPIO (GPIOx_MODER)

Register GPIOx_MODER, di mana x=A..I, digunakan untuk menentukan apakah sebuah pin I/O dijadikan sebagai masukan (*input*), keluaran (*output*), menjadi fungsi alternatif (terhubung ke periperal) atau dijadikan sebagai I/O analog (fungsi alternatif ADC atau DAC).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw										
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw										

Gambar 3.5 Register GPIOx_MODER

Setiap pin I/O diatur oleh masing-masing 2 bit register GPIOx_MODER, misalnya pin 0 akan diset melalui bit-0 dan bit-1, atau dalam register dinamakan dengan MODERO[1:0]. Berikut mode I/O yang bisa dipilih melalui register GPIOx_MODER:

1. MODERy[1:0] = 00, pin berfungsi sebagai input, (y=0..15)
2. MODERy[1:0] = 01, pin berfungsi sebagai output
3. MODERy[1:0]= 10, pin berfungsi sebagai fungsi alternatif (periperal)
4. MODERy[1:0]= 11, pin berfungsi di mode analog

Saat reset, nilai register GPIOx_MODER adalah 0x000 0000 artinya port akan berfungsi sebagai input, kecuali untuk PA (GPIOA_MODER) dan PB (GPIOB_MODER) yang saat reset beberapa pin I/Onya berfungsi sebagai port debug (JTAG).

3.2.2.2 Register Tipe Output GPIO (GPIOx_OTYPER)

Register GPIOx_OTYPER (*Output Type Register*) merupakan register untuk mengatur tipe output dari GPIO apakah *open drain* atau *push pull*. Register 32 bit ini hanya dipakai 15 bit yang mewakili tiap dari port yang bersangkutan, dan tiap bit diberi identitas OT0...OT15.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Gambar 3.6 Register GPIOx_OTYPER

Pengaturan bit OTy sebagai berikut:

1. Bit OTy = 0, pin yang bersangkutan akan di-set sebagai output push pull.
2. Bit OTy =1, pin yang bersangkutan akan di-set sebagai output open drain.

Ketika reset, register ini akan mempunyai nilai 0x0000 0000, artinya port akan di-set push pull. Register ini bisa dibaca dan ditulis (rw = *read write*).

3.2.2.3 Register Kecepatan Output GPIO (GPIOx_OSPEEDR)

Register ini digunakan untuk mengatur kecepatan GPIO. Dalam penggunaannya GPIO mungkin bisa dipakai untuk mengeluarkan clock pada frekuensi tinggi atau membaca sebuah sinyal clock frekuensi tinggi, salah satu contoh ketika sebuah GPIO difungsikan untuk fungsi alternatif misalnya untuk fungsi FSMC, USB atau ethernet yang semuanya memerlukan kecepatan tinggi.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]	OSPEEDR14 [1:0]	OSPEEDR13 [1:0]	OSPEEDR12 [1:0]	OSPEEDR11 [1:0]	OSPEEDR10 [1:0]	OSPEEDR9 [1:0]	OSPEEDR8 [1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR7[1:0]	OSPEEDR6[1:0]	OSPEEDR5[1:0]	OSPEEDR4[1:0]	OSPEEDR3[1:0]	OSPEEDR2[1:0]	OSPEEDR1 [1:0]	OSPEEDR0 [1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Gambar 3.7 Register GPIOx_OSPEEDR

Ada 4 kecepatan yang bisa dipilih berdasarkan bit-bit OSPEEDRy, di mana setiap pin I/O ditentukan oleh 2 bit OSPEEDRy, misal pin 0 dari sebuah Port ditentukan oleh bit OSPEEDR0[0] dan OSPEEDR0[1].

1. Kecepatan rendah, bit-bit OSPEEDRy[1:0] bernilai 00
2. Kecepatan menengah, bit-bit OSPEEDRy[1:0] bernilai 01
3. Kecepatan tinggi, bit-bit OSPEEDRy[1:0] bernilai 10
4. Kecepatan sangat tinggi, bit-bit OSPEEDRy[1:0] bernilai 11

Kecepatan GPIO juga dipengaruhi oleh tegangan catu, misal GPIO bisa mencapai frekuensi maksimum (120 MHz) pada tegangan catu minimal 2.7V dengan bit-bit OSPEEDRy diatur pada nilai 11. Karakteristik ini bisa dilihat di lembaran data ST32F207 (tabel 48). Selain itu perlu diperhatikan juga bahwa, men-set GPIO pada kecepatan maksimum bisa menimbulkan noise dan konsumsi daya lebih tinggi. Jadi kecepatan harus disesuaikan dengan penggunaanya, jika GPIO hanya dipakai untuk menyala LED misalnya, tidak perlu di-set di kecepatan tinggi.

3.2.2.4 Register Pull-Up/Pull-Down GPIO (GPIOx_PUPDR)

Register GPIOx_PUPDR digunakan untuk menentukan apakah sebuah pin GPIO terhubung dengan pull up atau pull down internal atau mengambang (floating). Pengaturan ini diperlukan misalnya dalam komunikasi I2C di mana pin data dan clock diharuskan terhubung ke sebuah resistor pull up. Dengan demikian pin GPIO yang digunakan untuk fungsi I2C bisa diatur ke kondisi mengambang dan menggunakan pull-up eksternal.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]	PUPDR14[1:0]	PUPDR13[1:0]	PUPDR12[1:0]	PUPDR11[1:0]	PUPDR10[1:0]	PUPDR9[1:0]	PUPDR8[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]	PUPDR6[1:0]	PUPDR5[1:0]	PUPDR4[1:0]	PUPDR3[1:0]	PUPDR2[1:0]	PUPDR1[1:0]	PUPDR0[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Gambar 3.8 Register GPIOx_PUPDR

Bit-bit PUPDRy[1:0] mengatur pengaturan pull up atau pull down sebagai berikut

PUPDRy[1:0] bernilai 00 sebuah pin I/O diset mengambang, tidak menggunakan pull up atau pull down

PUPDRAy[1:0] bernilai 01pin I/O diset untuk terhubung ke pull up

PUPDRy[1:0] bernilai 10, pin I/O diset untuk terhubung ke resistor pull down

4. PUPDRy[1:0] bernilai 11, tidak dipakai

3.2.2.5 Register Data GPIO (GPIOx_IDR dan GPIOx_ODR)

Register data ini digunakan untuk menulis atau membaca sebuah port GPIO. Untuk mengetahui kondisi sebuah port GPIO bisa dilakukan dengan membaca register GPIOx_IDR (*Input Data Register*), sedangkan untuk menentukan kondisi sebuah port bisa dilakukan dengan menulis ke register GPIOx_ODR (*Output Data Register*).

Register GPIOx_IDR hanya bisa dibaca dan hanya bisa diakses sebagai word (32 bit). Hanya bit ke-0 sampai ke-15 yang dipakai.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Gambar 3.9 Register GPIOx_IDR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Gambar 3.10 Register GPIOx_ODR

Register GPIOx_ODR akan mengatur pin ke-0 (bit ODR0) sampai pin ke-15 (bit ODR15) secara sekaligus, jika diinginkan menset/reset sebuah pin secara individual dilakukan melalui register GPIOx_BSRR.

3.2.2.6 Register Set/Reset Bit GPIO (GPIOx_BSRR)

Register GPIOx_BSRR (Bit Set Reset Register) digunakan untuk menentukan kondisi sebuah pin I/O secara individu tanpa mempengaruhi pin I/O yang lain. Berbeda dengan register GPIOx_ODR yang menentukan kondisi port secara paralel, register GPIOx_BSRR akan mengakses port secara bit. Walaupun demikian akses terhadap pin I/O tetap dilakukan melalui register ODR artinya penulisan BSRR akan mempengaruhi bit ODR yang bersangkutan.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Gambar 3.11 Register GPIOx_BSRR

Register BSRR merupakan register yang hanya bisa ditulis, pembacaan terhadap register ini akan selalu menghasilkan 0x0000. Register ini bisa diakses sebagai word, half-word ataupun byte. Bit-0 sampai bit-15 merupakan bit BSy (bit untuk menset pin I/O) dan bit-16 sampai bit-31 merupakan bit BRy (bit untuk mereset pin IO). Menset bit BSy akan menset bit ODR yang bersangkutan sekaligus menset pin IOnya. Dan menset bit BRy akan mereset bit ODR dan mereset pin IOnya. Jika bit BSy

dan BRy keduanya diset, maka bit BSy akan diprioritaskan artinya akan menset pin IO.

3.2.2.7 Register Pengunci Konfigurasi GPIO (GPIOx_LCKR)

Register LCKR digunakan untuk mengunci konfigurasi sebuah pin IO sehingga konfigurasi tidak bisa diubah kecuali jika diinginkan. Proses penguncian dilakukan dengan melakukan 3 kali penulisan secara berurutan terhadap bit-16 (LCKK) sementara bit-0 sampai bit-15 yang menentukan apakah penguncian dilakukan atau tidak. Pada saat proses penulisan bit-0 sampai bit-15 tidak boleh berubah.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
															LCKK
															rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Gambar 3.12 Register GPIOx_LCKR

Berikut cara penguncian konfigurasi sebuah port:

1. Tentukan pin mana saja yang akan dikunci dengan menset bit LCKnya.
2. Set bit LCKK, bit-bit LCK tidak boleh berubah
3. Reset bit LCKK,
4. Set kembali bit LCKK
5. Baca bit LCKK (harus bernilai 1) hanya untuk memastikan bahwa proses penguncian berhasil.

Pada saat proses penulisan bit LCK[15:0] tidak boleh berubah. Setelah proses penguncian, konfigurasi port tidak akan berubah sampai CPU mereset.

3.2.2.8 Register Fungsi Alternatif

Seperti telah dijelaskan sebelumnya, pin I/O bisa difungsikan sebagai pin yang terhubung ke periperal internal (fungsi alternatif) melalui register GPIOx_MODER. Periperal mana saja yang terhubung ke pin I/O telah ditentukan pada saat perancangan dan produksi STM32F207. Sebuah pin I/O bisa terhubung ke beberapa periperal dan periperal mana yang akan terhubung ditentukan oleh register fungsi alternatif GPIOx_AFRL dan GPIOx_AFRH. Register GPIOx_AFRL (*Alternative Function Register*

Low) untuk mengatur fungsi alternatif pin 0 - pin 7 sedangkan GPIOx_AFRH (*Alternative Function Register High*) untuk mengatur pin 8 - pin 15.

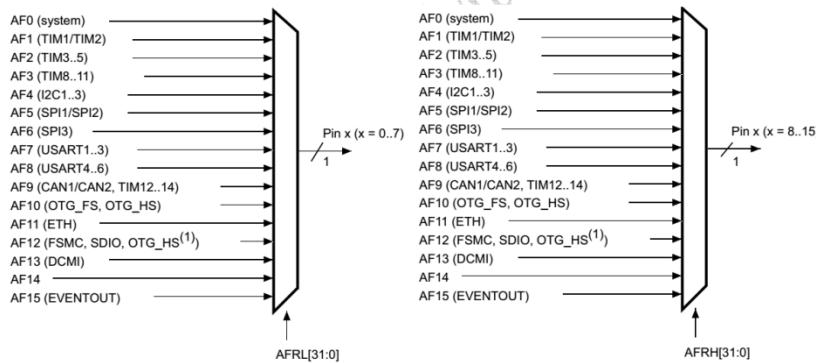
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw												

Gambar 3.13 Register GPIOx_AFRL

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
rw	rw	rw	rw												
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
rw	rw	rw	rw												

Gambar 3.14 Register GPIOx_AFRH

Fungsi alternatif setiap pin diatur oleh 4 bit register GPIOx_AFRL atau GPIOx_AFRH, sehingga memungkinkan sebuah pin bisa difungsikan menjadi fungsi alternatif 0 (AF0) sampai fungsi alternatif 15 (AF15) di mana nilai AFRLy atau AFRHy dari 0000 - 1111 (dalam biner). Untuk keterangan lebih detail mengenai fungsi AF0 - AF15 masing-masing port bisa dilihat di datasheet STM32F207 (Tabel 10 halaman 60).



Gambar 3.15 Fungsi Alternatif AF0 - AF15

3.2.2.9 Langkah-Langkah Pengaturan GPIO

Sebelum difungsikan, sebagai GPIO atau fungsi alternatif, sebuah port harus dilakukan pengaturan dulu melalui register-register pengaturnya

1. Sebagai GPIO, pilih mode melalui register GPIOx_MODER sebagai input atau output.
2. Sebagai fungsi alternatif, untuk fungsi analog (ADC atau DAC), pilih mode analog melalui register GPIOx_MODER, selain itu pilih mode fungsi alternatif. Kemudian pilih salah satu fungsi alternatif (AF0 - AF15) melalui register GPIOx_AFRL atau GPIOx_AFRH.
3. Tentukan tipe output, pull up/pull down dan kecepatan melalui register masing-masing.
4. Sebagai GPIO, data dapat dibaca atau ditulis ke IO melalui register IDR dan ODR

Seperti yang telah disebutkan, beberapa pin I/O tidak berada di mode input sesaat setelah reset. Pin-pin ini berada di mode fungsi alternatif AF0 (sistem), yaitu terhubung ke port debug. Port debug ini akan menghubungkan software debugger ke core Cortex-M3. Tabel 3.3 menunjukkan pin-pin tersebut.

Tabel 3.3 Pin-Pin Debug

Pin	Fungsi Debug		Keterangan
	JTAG	SWDIO	
PA13	JTMS	SWDIO	JTAG Test Mode Selection Serial Wire Data Input/Output
PA14	JTCK	SWCLK	JTAG Test Clock Serial Wires Clock
PA15	JTDI		JTAG Test Data Input
PB3	JTDO	TRACESWO	JTAG Test Data Output TRACE jika difungsikan
PB4	NJTRST		JTAG Test nReset

Tetapi pin-pin tersebut tetap bisa difungsikan sebagai pin I/O biasa. Walaupun demikian selama pengembangan tetap disarankan untuk selalu mengaktifkan port debug. ARM Cortex-M3 mempunyai port debug melalui mode JTAG dan SWD. Mode JTAG memerlukan lebih banyak pin (5 pin) dibandingkan dengan SWD (2 pin) sehingga fungsi JTAG bisa dimatikan dengan tetap mengaktifkan mode SWD untuk

keperluan debugging. Cara penonaktifannya sama dengan pengaturan GPIO di atas. Jika pin JTAG akan difungsikan sebagai IO biasa, mode kerja harus diatur ke mode input atau output karena fungsi JTAG adalah fungsi altenatif (AF0) melalui register GPIOx_MODER. Dan jika pin JTAG mau difungsikan sebagai fungsi alternatif selain JTAG tinggal diubah melalui register GPIOx_AFRL atau GPIOx_AFRH.

Selain itu, ada juga pin I/O yang dipakai untuk koneksi kristal atau eksternal osilator, pin-pin tersebut adalah PC14 dan PC15 untuk koneksi ke OSC32_IN dan OSC32_OUT (kristal frekuensi rendah) dan PH0 dan PH1 untuk koneksi ke OSC_IN dan OSC_OUT (kristal frekuensi tinggi). Sesaat setelah reset STM32F207 akan bekerja dengan osilator internal pada saat ini pin-pin tersebut masih bekerja sebagai GPIO, jika kemudian diperlukan bekerja dengan dengan kristal eksternal, maka bit LSEON (untuk frekuensi rendah) atau bit HSEON (untuk frekuensi tinggi) di register RCC_BDCR (akan dijelaskan kemudian) harus diset. Setelah di-set fungsi GPIO untuk pin-pin tersebut hilang (tidak bisa dijadikan fungsi GPIO atau fungsi alternatif lain).

3.3 TIMER

STM32F207 mempunyai 14 buah timer yang diberi nama TIM1 - TIM14. Timer-timer ini berdasarkan fungsi dan fitur dikelompokan menjadi 3, yaitu

1. Timer dasar (*basic timer*),
2. Timer fungsi umum (*general purpose timer*) dan
3. Timer tingkat lanjut (*advanced timer*).

Timer dasar merupakan timer yang hanya mempunyai fungsi dasar pewaktuan (*time base*) dan tambahan fungsi untuk menyulut (triger) DAC. Timer ini tidak mempunyai input atau output. Timer fungsi umum merupakan timer yang selain menyediakan fungsi pewaktuan juga mempunyai input dan output. Inputnya bisa digunakan untuk pengukuran frekuensi sedangkan outputnya bisa digunakan misal untuk mengeluarkan PWM. Sedangkan timer fungsi lanjut merupakan timer dengan fitur-fitur tambahan secara hardware misal untuk pengendalian motor.

Tabel 3.4 Timer-timer STM32F207

Jenis Timer	Nama Timer	Lebar Pencacah
-------------	------------	----------------

Timer Dasar	TIM6 dan TIM7	16 bit
Timer Fungsi Umum	TIM3, TIM4, TIM9, TIM10, TIM11, TIM12, TIM13, dan TIM14	16 bit
	TIM2 dan TIM5	32 bit
Timer Tingkat Lanjut	TIM1 dan TIM8	16 bit

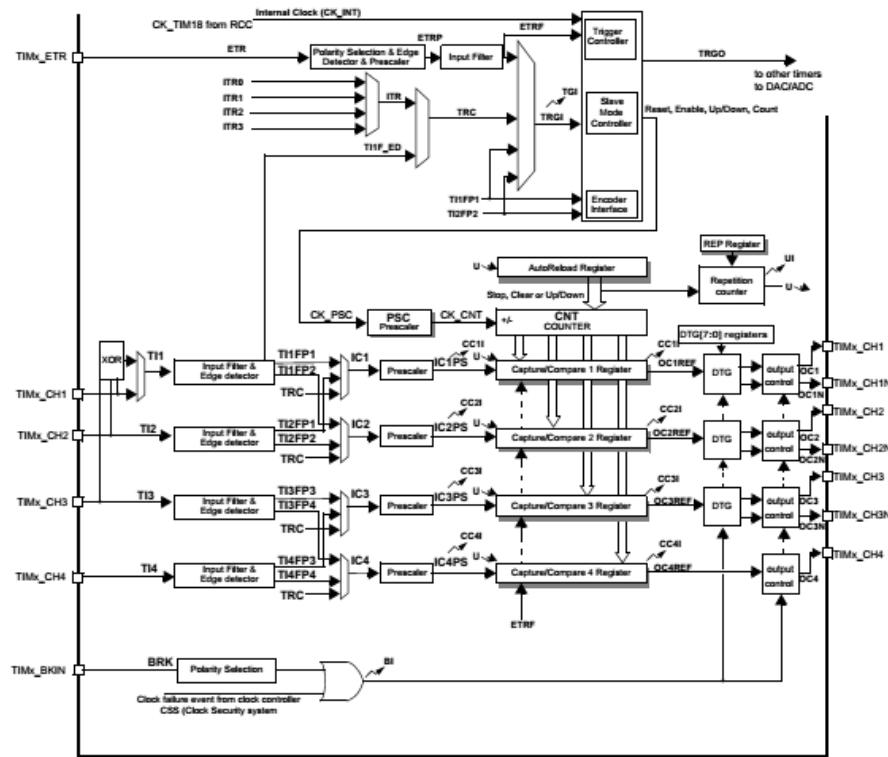
Timer-timer tersebut bekerja secara mandiri dan mempunyai sumber (register data dan kendali, vektor interupsi) masing-masing. Tetapi timer-timer tersebut bisa diatur untuk bekerja secara sinkron sehingga bisa membentuk sebuah rangkaian timer dengan fungsi komplek tanpa memerlukan software yang rumit. Selain itu semua timer bisa dihubungkan dengan DMA.

Timer mempunyai struktur, seperti ditunjukkan oleh diagram blok TIM1, sebagaimana berikut:

1. Unit kendali master/slave
2. Unit *time-base*
3. Unit *timer-channel*
4. Unit fitur penggereman (*break feature unit*)

Unit kendali master/slave berfungsi terutama untuk pengendalian unit time-base seperti mengatur sinyal clock dan arah pencacahan (naik atau turun) dari unit time-base. Pada fungsi sinkronisasi timer, unit ini akan menentukan apakah timer berfungsi sebagai master atau slave. Unit ini juga bisa digunakan untuk mengeluarkan sinyal sinkronisasi (sinyal TRGO) ke timer lain atau ke ADC/DAC. Tidak semua timer STM32F207 mempunyai unit kendali master/slave yang berfungsi penuh misalnya timer dasar (TIM6 dan TIM7).

Unit time-base merupakan bagian timer yang akan melakukan pencacahan (counting) sinyal clock yang diumpulkan dari unit master/slave. Secara umum timer akan mempunyai sebuah pencacah (*counter*) 16 bit atau 32 bit. Pencacah tersebut bisa diprogram untuk melakukan pencacahan naik (*up counter*), pencacah turun (*down counter*) atau bahkan naik dan turun (*up/down counter*).



Gambar 3.16 Diagram Blok TIM1 dan TIM8

Pencacahan dilakukan dari sumber clock yang bisa dipilih dan nilai pembagi clock (*pre-scaler*) yang bisa diprogram. Pada saat jadi pencacah naik, nilai cacahan akan berhenti di sebuah nilai *auto-reload* sedangkan saat jadi pencacah turun cacahan di mulai dari nilai auto-reload. Nilai auto-reload ini bisa diprogram dan bisa diperbaharui walau pun timer sedang jalan (*on the fly*). Setiap akhir cacahan, timer bisa diprogram untuk membangkitkan interupsi atau event. Timer juga bisa mempunyai input atau output yang bisa dihubungkan ke salah satu GPIO, sehingga bisa digunakan untuk mengukur frekuensi sebuah sinyal yang berasal dari sumber luar atau mengeluarkan sinyal dengan frekuensi dan lebar pulsa yang bisa diprogram.

Pada timer tingkat lanjut unit time-base dilengkapi juga dengan pencacah pengulangan (repetition counter). Pencacah ini akan mencacah sudah berapa kali kondisi overflow atau underflow terjadi. Setiap kali terjadi overflow atau underflow, isi pencacah pengulangan akan dikurangi 1 dan

akan membangkitkan event jika isi register mencapai nol. Isi register pengulangan akan diisi ulang (auto-reload) dengan nilai yang tersimpan di register TIMx_RCR. Dengan adanya pencacah pengulangan ini, timer hanya akan membangkitkan event saat pencacah pengulangan mencapai nol bukan saat terjadinya overflow atau underflow. Pencacah pengulangan ini berguna terutama saat pembangkitan sinyal PWM.

Unit timer-channel merupakan bagian timer yang berfungsi untuk menghubungkan dengan eksternal mikrokontroler, sehingga unit ini akan dipetakan ke pin mikrokontroler. Unit ini bisa diatur sebagai masukan untuk menerima sinyal clock dari luar atau sebagai keluaran untuk mengeluarkan sinyal PWM misalnya.

Unit fitur penggereman (*break feature unit*) hanya ada di timer tingkat lanjut (TIM1 dan TIM8). Fungsi ini mengharuskan timer mempunyai keluaran yang komplemen. Unit ini bisa digunakan pemadaman yang aman pada inverter daya.

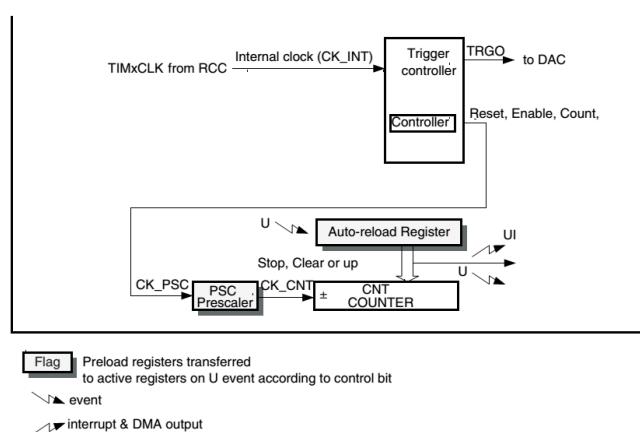
3.3.1 FITUR UTAMA

Fitur lengkap timer STM32F207 adalah sebagai berikut:

1. Mempunyai pencacah isi ulang (auto-reload counter) yang bisa diprogram untuk mencacah naik (up counter), turun (down counter) atau naik/turun. Semua timer mempunyai pencacah 16 bit, kecuali TIM2 dan TIM5 yang mempunyai pencacah 32 bit.
2. Mempunyai register pra-skalar 16 bit yang bisa digunakan untuk membagi sinyal clock masukan dengan 1 - 65536
3. Mempunyai sampai 4 kanal input capture, output compare, pembangkitan sinyal PWM dan mode keluaran 1 pulsa.
4. Keluaran komplemen dengan fitur dead time yang dapat diprogram
5. Rangkaian sinkronisasi untuk mengendalikan sinyal eksternal dan sinkronisasi dengan beberapa timer lain
6. Dilengkapi dengan pencacah pengulangan sehingga timer hanya akan membangkitkan sebuah event setelah beberapa kali terjadi siklus di pencacah timer.
7. Masukan penggereman (break input) untuk membuat keluaran timer berada di kondisi reset atau kondisi yang diketahui, hanya ada di timer tingkat lanjut.

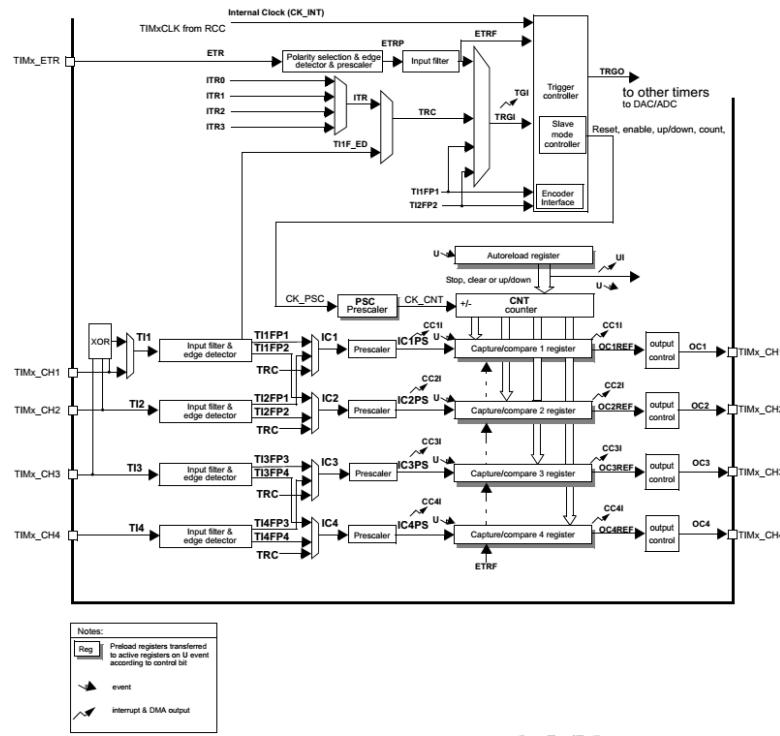
8. Pembangkitan interupsi atau DMA saat pencacahan mengalami overflow, underflow, inisialisasi atau pencacahan oleh pemicu dari internal atau eksternal.
9. Mendukung enkoder *quadrature/incremental* maupun sensor *hall* untuk keperluan pengukuran atau penentuan posisi.
10. Masukan trigger untuk clock eksternal atau manajemen arus siklus demi siklus.

Fitur-fitur lengkap tersebut hanya dimiliki oleh timer tingakt lanjut (TIM1 dan TIM8), sementara timer dasar dan timer fungsi umum hanya mempunyai sebagian fitur tersebut. Sebagai contoh timer dasar tidak mempunyai input capture maupun output compare sehingga timer dasar tidak dipetakan ke pin mikrokontroler, tetapi timer dasar secara khusus difungsikan untuk memicu kerja DAC yang tidak dimiliki oleh timer lain.



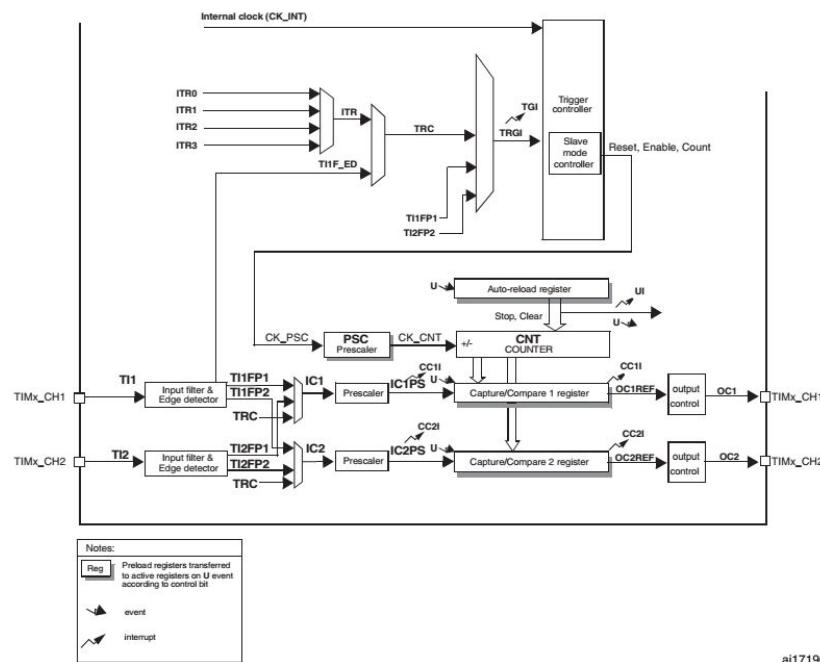
Gambar 3.17 Diagram Blok Timer Dasar

Timer dasar dan juga timer fungsi umum (TIM9 - TIM14) hanya bisa melakukan pencacahan naik (*count up*), sementara timer lain bisa juga diprogram untuk melakukan cacahan turun (*count down*) atau cacahan naik/turun (*count up/down*).



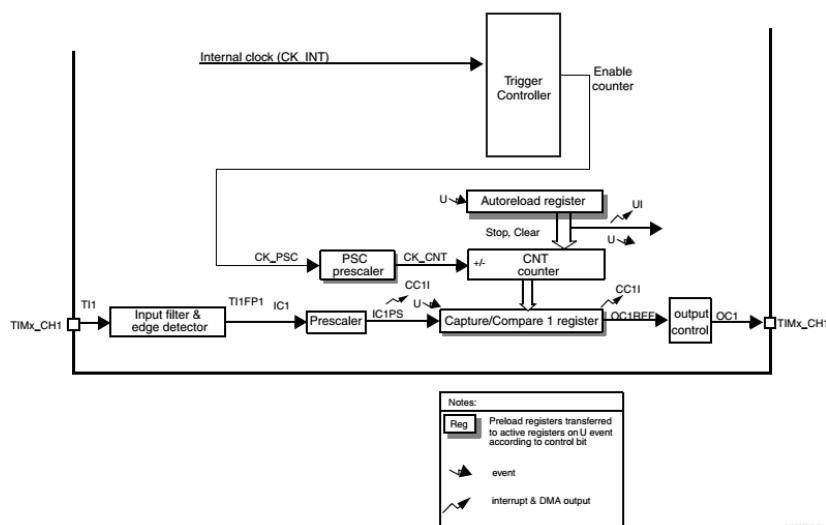
Gambar 3.18 Diagram Blok TIM2 - TIM5

Timer tingkat lanjut (TIM1 dan TIM8) serta timer fungsi umum (TIM2 - TIM5) masing-masing mempunyai sampai 4 input *capture* dan 4 output *compare* (TIMx_CH1 - TIMx_CH4) dan sebuah masukan pemicu eksternal (TIMx_ETR). Sedangkan TIM9 dan TIM12 masing-masing mempunyai 2 input capture dan 2 output capture, sementara di TIM10, TIM11, TIM13 dan TIM14 hanya mempunyai 1 input capture dan 1 output compare. Input capture bisa digunakan untuk pengukuran frekuensi sebuah sinyal atau membaca sebuah enkoder atau sensor *hall* untuk pengukuran posisi. Sedangkan output compare bisa digunakan untuk mengeluarkan sinyal misal sinyal PWM untuk pengendalian motor.



ai17190

Gambar 3.19 Diagram Blok TIM9 dan TIM12



ai17725c

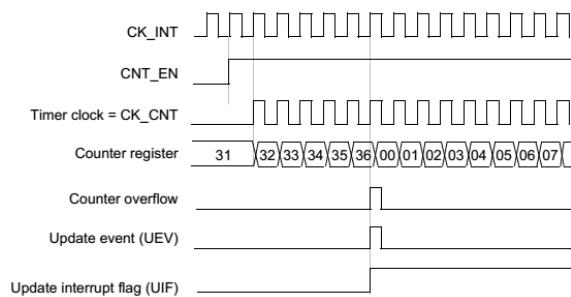
Gambar 3.20 Diagram Blok TIM10/11/13/14

3.3.2 MODE KERJA TIMER

3.3.2.1 Mode Cacahan

Timer bisa diprogram untuk melakukan cacahan naik (*up counter*), cacahan turun (*down counter*) dan cacahan naik/turun atau *central-aligned counter*, kecuali TIM9 - TIM14 dan timer dasar yang hanya bisa melakukan cacahan naik.

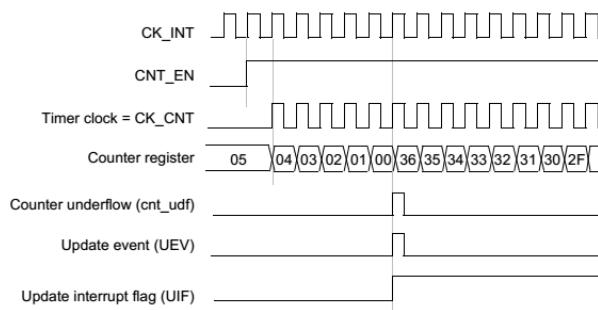
Pada mode cacahan naik, timer/counter akan melakukan cacahan dimulai dari nol kemudian naik sampai mencapai nilai auto reload. Pada saat ini timer mengalami overflow dan bisa diprogram untuk membangkitkan interupsi atau memicu DMA. Cacahan kemudian diawali lagi dari nol.



Gambar 3.21 Cacahan Naik dengan Nilai Auto-reload 36

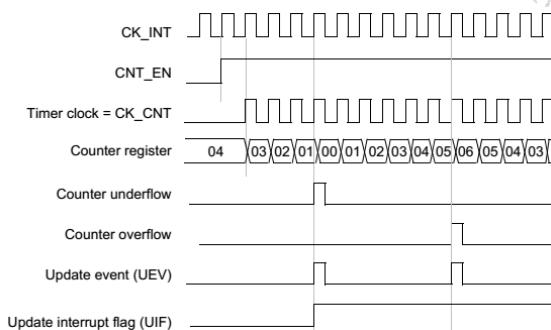
Gambar 3.21 menunjukkan timer saat cacahan naik, sumber clock diambil dari sumber internal (CK_INT). Dengan nilai pembagi clock (pra-skalar) 1, sehingga clock timer (CK_CNT) akan sama dengan CK_INT. Cacahan dimulai saat CNT_EN di-set, cacahan dimulai dari 0 dan naik sampai nilai auto-reload (36), saat mencapai nilai auto-reload, timer mengalami overflow dan akan men-set *update event* (UEV) dan flag interupsi (UIF). Cacahan kemudian diawali lagi dari 0.

Saat mode cacahan turun, timer akan mulai mencacah dari nilai auto reload kemudian turun sampai mencapai nol. Timer dikatakan mengalami underflow dan bisa membangkitkan interupsi atau memicu DMA. Cacahan kemudian akan diawali lagi dari nilai auto reload.



Gambar 3.22 Cacahan Turun dengan Nilai Auto-reload 36

Pada mode central aligned, timer akan memulai pencacahan dari nol dan naik sampai mencapai nilai auto reload - 1. Pada saat ini timer bisa membangkitkan interupsi overflow. Timer kemudian melanjutkan dengan melakukan cacahan turun dimulai dari nilai auto reload dan turun sampai mencapai 1. Interupsi underflow bisa dibangkitkan. Dan timer akan kembali mencacah naik dari 0.



Gambar 3.23 Mode Central-aligned Counter dengan Nilai Auto-reload 36

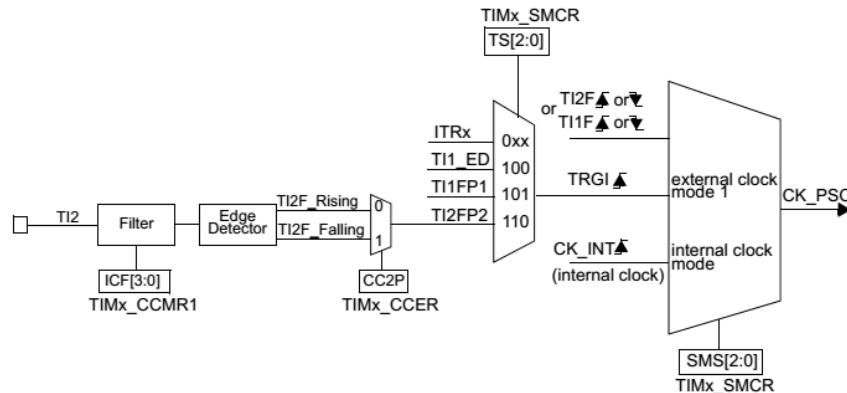
3.3.2.2 Sumber Clock

Sumber clock bisa dipilih sebagai berikut:

1. Clock internal (C_INT).
2. Clock eksternal mode 1 (TIx).
3. Clock eksternal mode 2 (ETR), hanya untuk TIM1, TIM2, TIM3, TIM4 dan TIM8 .
4. Picu (trigger) internal (ITRx), kecuali TIM6/7/10/11/13/14.

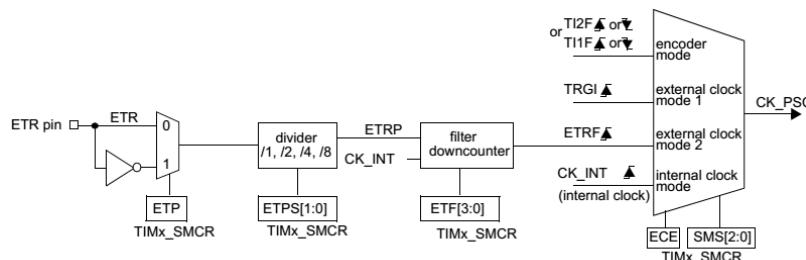
Sumber clock ini kemudian akan dibagi oleh pra-skalar sebelum masuk ke pencacahan. Semua timer mempunyai pra-skalar 16 bit.

Clock eksternal mode 1 diumpulkan melalui pin TIx (TIMx_CHx). Sumber clock eksternal ini bisa diatur untuk di-filter atau tidak. Counter bisa melakukan pencacahan saat transisi naik (*rising*) atau transisi turun (*falling*) dari sinyal clock tersebut.



Gambar 3.24 Clock Eksternal Mode 1

Clock eksternal mode 2 diumpulkan ke pin pemicu eksternal (TIMx_ETR). Clock eksternal ini bisa dibagi 1, 2, 4, atau 8. Oleh karena ada pembagi di sisi masukannya, frekuensi clock mode 2 ini bisa lebih tinggi dari pada frekuensi internal timer, misal frekuensi clock bus APB. Clock eksternal mode 2 hanya dimiliki oleh TIM1, TIM2, TIM3, TIM4 dan TIM8. Clock mode 2 hanya bisa melakukan pencacahan di sisi naik sinyal clock.



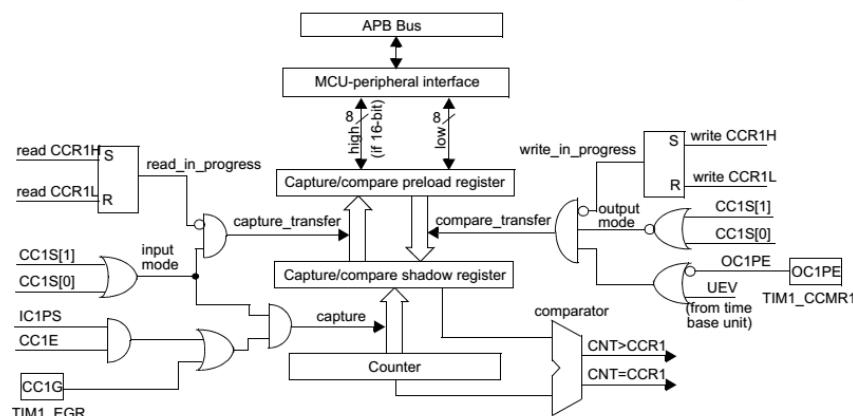
Gambar 3.25 Clock Eksternal Mode 2

Fungsi filter di sisi masukan adalah untuk memvalidasi sinyal clock sesuai dengan durasi yang telah ditentukan. Jika durasi atau periode sinyal clock kurang dari durasi ambang yang telah ditentukan, sinyal tersebut tidak akan dicacah sebagai 1 perioda.

Mode picu internal, clock berasal dari output timer lain, sehingga beberapa timer bisa bekerja secara sinkron.

3.3.2.3 Unit Capture dan Compare

Setiap timer, kecuali timer dasar mempunyai unit *capture* dan *compare*. Unit ini dikendalikan oleh sebuah register. Register ini mempunyai fungsi yang berbeda tergantung dari pengaturan, jika sebagai capture, register ini akan menjadi input dengan filter dan mode pengukuran PWM (*Pulse Width Modulation*) sedangkan sebagai compare register ini akan berfungsi sebagai pembanding dan fungsi untuk menghasilkan sinyal PWM dan mode 1 pulsa.

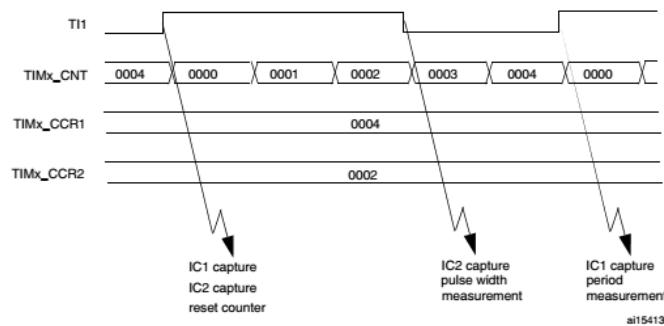


Gambar 3.26 Unit Capture/Compare

Unit capture dan compare ini sebenarnya terdiri atas sebuah register *preload* dan register *shadow*. Tetapi pembacaan dan penulisan ke unit ini akan selalu melalui register *preload*. Pada mode capture, data hasil capture akan disimpan ke register *shadow* yang kemudian disimpan juga ke register *preload*. Pada mode compare, isi register *preload* akan digandakan ke register *compare* untuk kemudian dibandingkan dengan pencacah.

3.3.2.4 Mode Input PWM

Mode ini digunakan untuk mengukur frekuensi maupun siklus kerja (*duty cycle*) sebuah sinyal PWM dari luar. Memanfaatkan mode input capture. Pada mode ini, 2 buah masukan (misal IC1 dan IC2) dan 2 buah register capture digunakan, misal TIMx_CCR1 dan TIMx_CCR2. Kedua masukan ini diprogram untuk menggunakan pin input timer yang sama misal TI1 tetapi kedua unit capture diprogram untuk bekerja pada transisi sinyal yang berlawanan, yang satu aktif saat transisi naik (IC1) sedangkan register capture yang lainnya aktif saat transisi turun (IC2).



Gambar 3.27 Pengukuran Sinyal PWM

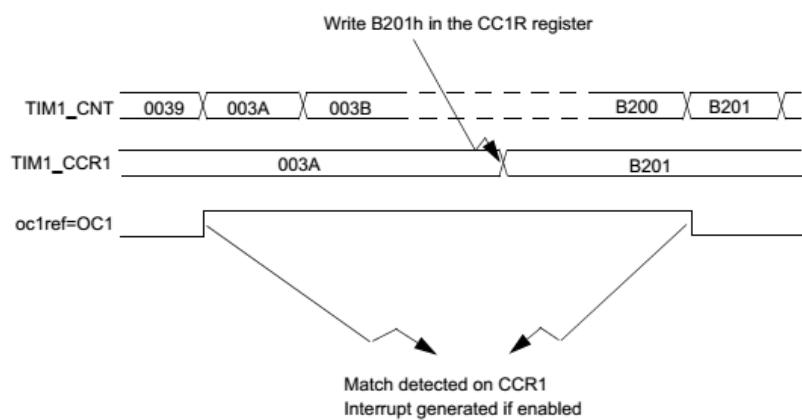
Kendali slave diprogram di mode reset. Pada mode ini, sinyal transisi naik akan menginisialisasi atau mereset pencacah. Pencacah akan mulai melakukan cacahan dari awal. Pada saat transisi turun, akan terjadi capture di IC2, sehingga isi pencacah akan disimpan ke register TIMx_CCR2. Kemudian pada transisi naik berikutnya terjadi capture di IC1 dan isi pencacah akan disimpan di register TIMx_CCR1 dan pencacah pun akan mengalami reset.

Dengan memperhatikan proses di atas, TIMx_CCR2 akan menyimpan periode saat sinyal clock berada pada logika tinggi, dengan kata lain menyimpan lebar pulsa (*duty cycle*) dan TIMx_CCR1 akan menyimpan periode sinyal yang diukur.

3.3.2.5 Mode Output Compare dan PWM

Unit compare bisa digunakan untuk mengeluarkan sinyal atau pun sebagai penanda bahwa sebuah periode waktu telah berjalan. Unit compare ini akan men-set pin yang dipetakan ke timer sebagai output.

Sesuai dengan namanya unit ini akan membandingkan isi pencacah dengan isi TIMx_CCR_x , ketika isinya sama (*match*) maka output timer (OC_x) bisa diatur untuk di-set, reset atau *toggle*. OC_x ini bisa dihubungkan dengan pin mikrokontroler, sehingga kondisi di pin tersebut akan sama dengan kondisi OC_x . Pada saat match juga timer bisa diatur untuk mengirim sinyal interupsi ke CPU.



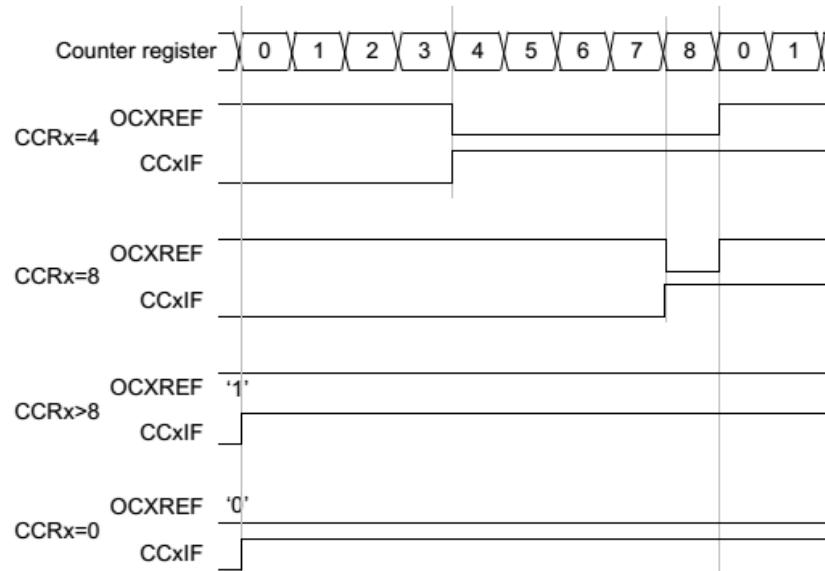
Gambar 3.28 Mode Output Compare

Salah satu contoh penggunaan unit compare adalah pembangkitan sinyal PWM (*Pulse Width Modulation*). Sinyal PWM bisa dihasilkan dengan frekuensi ditentukan oleh register auto-reload (TIMx_ARR) dan siklus kerja (*duty cycle*) ditentukan dengan nilai yang ada di register TIMx_CCR_x .

Sinyal PWM bisa diatur secara mandiri untuk setiap unit compare (satu sinyal PWM untuk setiap output timer OC). Ada 2 mode PWM yang bisa dipilih mode 1 dan mode 2. Di mode PWM mode 1, sinyal OC_xREF akan aktif (logika 1), selama nilai pencacah $\text{TIMx_CNT} < \text{TIMx_CCR}_x$ dan menjadi tidak aktif ($\text{OC}_x\text{REF}=0$) selama $\text{TIMx_CNT} > \text{TIMx_CCR}_x$. Sedangkan di PWM mode 2, sinyal OC_xREF akan menjadi aktif selama $\text{TIMx_CNT} > \text{TIMx_CCR}_x$ dan menjadi tidak aktif saat $\text{TIMx_CNT} < \text{TIMx_CCR}_x$. Sinyal OC_xREF ini akan menjadi referensi untuk sinyal OC, artinya saat OC_xREF aktif sinyal OC juga akan aktif. Logika aktif untuk sinyal OC bisa diatur melalui bit CCxP di register TIMx_CCER apakah aktif tinggi atau aktif high.

Berdasarkan arah pencacahan, PWM bisa dibangkitkan sebagai mode *edge-aligned* saat pencacahan diatur untuk pencacahan naik atau pencacahan turun dan mode *center-aligned*.

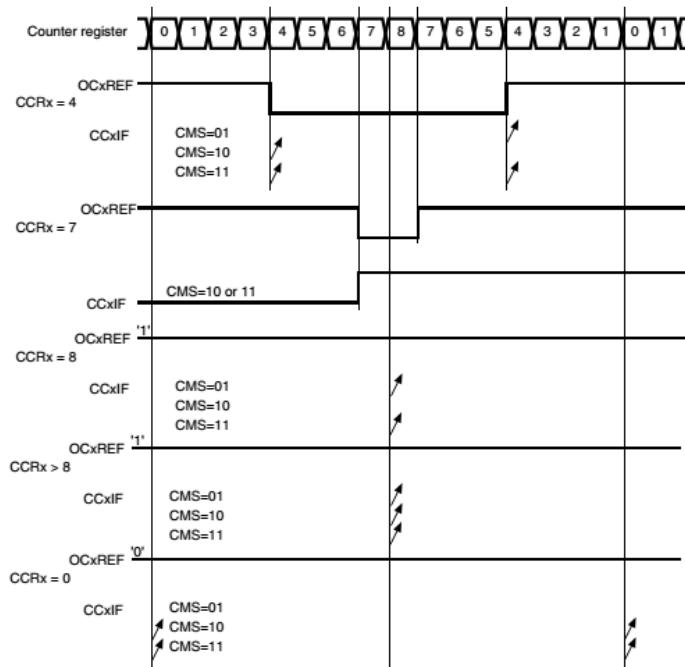
Di mode *edge-aligned*, pencacah diatur untuk mencacah naik atau turun. Gambar 3.29 menunjukkan diagram pewaktuan PWM *edge-aligned* pada saat pencacahan naik. Nilai auto-reload adalah 8 dan PWM mode 1. Selama nilai register pencacah $\text{TIMx_CNT} < \text{TIMx_CCRx}$, OCxREF akan aktif (berlogika 1) dan menjadi tidak aktif saat $\text{TIMx_CNT} > \text{TIMx_CCRx}$. Ketika nilai compare (CCRx) lebih besar dari nilai auto reload, OCxREF akan menjadi 1 dan ketika $\text{CCRx}=0$ maka OCxREF akan menjadi 0.



Gambar 3.29 PWM Edge-aligned

Saat pencacahan turun, OCxREF akan aktif selama $\text{TIMx_CNT} > \text{TIMx_CCRx}$. Ketika nilai CCRx lebih besar dari pada nilai auto-reload, OCxREF akan menjadi 1. Dan karena nilai pencacah akan selalu lebih besar dari pada nilai CCRx , di mode ini tidak bisa membangkitkan 0% PWM.

Di mode PWM center-aligned, pencacah diatur untuk melakukan pencacahan naik dan turun. Konsisi aktif atau tidaknya OCxREF tetap tergantung mode 1 atau mode 2. Gambar 3.30 menunjukkan diagram pewaktuan PWM center-aligned.



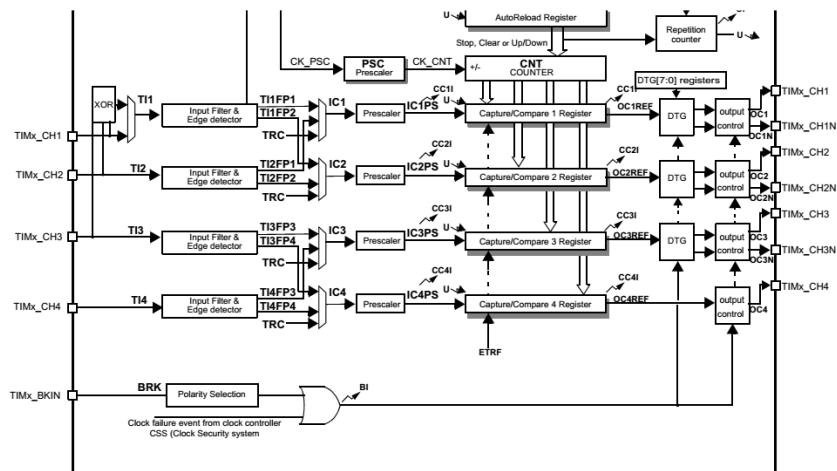
Gambar 3.30 PWM Center-aligned

3.3.2.6 Fungsi Break

Unit break, yang hanya dimiliki oleh TIM1 dan TIM8, digunakan untuk pengendalian motor atau sistem catu daya menggunakan sinyal PWM, terutama untuk sistem proteksi. Jika terjadi masalah di sistem kendali, unit break ini bisa diatur agar output timer diset di kondisi yang aman, menghentikan sinyal PWM. Sinyal break bisa berasal dari luar melalui pin TIMx_BKIN (sinyal BRK) dan dari internal melalui sistem keamanan clock (*Clock Security System*) yang berasal dari kendali clock reset (*Reset Clock Controller*) yang dinamakan dengan sinyal BRK_ACTH.

Kedua sinyal break ini di-OR-kan dan digunakan untuk mengendalikan sinyal OCxREF dan output compare (OC) dan juga keluaran komplemennya. Setelah reset unit break berada dikondisi tidak aktif. Fungsi break bisa diaktifkan melalui bit BKE di register TIMx_BDTR. Sedangkan polaritas dari sinyal break, apakah aktif tinggi atau aktif rendah, diset melalui bit BKP di register yang sama. Ketika sinyal

BRK_ACTH diaktifkan sebagai sumber break, polaritasnya harus diset aktif tinggi. Kalau tidak, sinyal PWM tidak akan dibangkitkan secara mandiri dari sinyal break yang berasal dari sumber internal.



Gambar 3.31 Unit Break

Di dalam aplikasi, pin TIMx_BKIN bisa dihubungkan dengan sensor tegangan atau sensor arus yang bisa berupa sebuah rangkaian pembanding (comparator). Keluaran pembanding disesuaikan dengan polaritas sinyal BRK. Jika sensor mendeteksi adanya masalah, misal arus atau tegangan berlebih, sensor segera mengirimkan sinyal break ke mikrokontroler, dan unit break dengan segera menghentikan keluaran PWM. Mikrokontroler bisa diprogram juga untuk memberikan indikasi error melalui LED, buzzer atau perangkat tampilan lainnya.

3.3.2.7 Antarmuka Rotary Encoder dan Sensor Efek Hall

Rotary encoder merupakan perangkat elektro-mekanik yang mengubah posisi atau gerakan putaran ke dalam bentuk sinyal analog atau sinyal digital. Rotari enkoder banyak ditemukan di sistem kendali industri, misal untuk pengukuran kecepatan atau posisi putaran sebuah motor listrik. Rotari enkoder juga banyak dipakai diperangkat audio mobil sebagai pengganti potensiometer analog dan dipakai juga sebagai pendeksi posisi di mouse komputer ketika masih menggunakan teknologi opto-mekanikal.

Ada 2 jenis rotari enkoder:

1. Jenis absolut
2. Jenis penambahan (*incremental*)

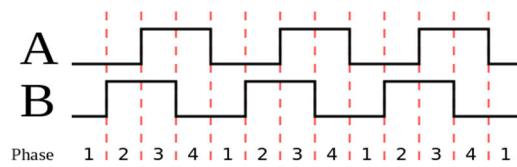
Jenis absolut biasanya mempertahankan posisi terakhirnya walaupun catu daya dihilangkan. Outputnya biasanya dalam bentuk data paralel, misal 3 bit data.

Jenis penambahan memberikan data saat terjadi perubahan posisi. Diperlukan perangkat lain untuk menentukan posisi sebenarnya, misal dengan menggunakan mikrokontroler. Enkoder jenis penambahan ini biasanya mempunyai 2 bit output, sinyal A dan sinyal B. Kedua sinyal ini memiliki beda fase sebesar 90°, oleh karena itu biasa disebut dengan sinyal quadrature. Enkoder jenis inilah yang bisa ditangani secara langsung oleh timer STM32F207.



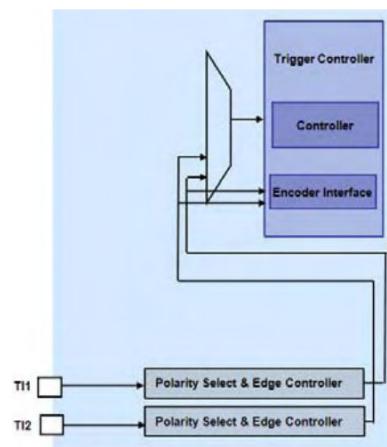
Gambar 3.32 Contoh Rotary Enkoder Jenis Penambahan

Dengan memperhatikan keluaran dari rotari enkoder ini maka bisa ditentukan arah putaran dari enkoder, apakah enkoder diputar searah atau berlawanan dengan jarum jam. Misal prosesor membaca keluaran enkoder (00 → 01 → 11) maka enkoder diputar searah jarum jam, dan ketika keluaran enkoder (00 → 10 → 11) maka enkoder diputar berlawanan arah dengan jarum jam. Selain itu karena enkoder merupakan perangkat elektro-mekanis, maka bisa menghasilkan efek *bouncing* yang bisa membuat pembacaan salah. Sehingga diperlukan penanganan terhadap efek ini.



Gambar 3.33 Bentuk Gelombang Enkoder Jenis Penambahan

Untuk membaca rotary enkoder digunakan 2 input timer (T1 dan T2). Polaritas kedua input bisa diprogram apakah aktif saat transisi naik, transisi turun atau keduanya. Filter juga bisa diaktifkan untuk mengurangi efek bouncing dari enkoder. Pencacah bisa diprogram untuk mencacah saat T1 aktif, T2 aktif atau saat keduanya aktif. Nilai cacahan bisa naik atau turun sesuai urutan bit yang diterima dari pin T1 dan T2. Nilai cacahan maksimal ditentukan oleh nilai register auto-reload (TIMx_ARR). Selain itu bit DIR di register TIMx_CR1 bisa di-set atau di-reset sesuai dengan arah cacahan.

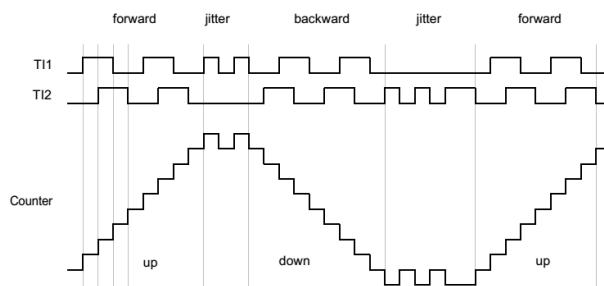


Gambar 3.34 Antarmuka Rotary Enkoder dengan Timer

Tabel 3.5 Sinyal Enkoder dan Arah Cacahan

Sinyal Aktif	Level di pin Masukan yang berlawanan	TI1FP1		TI2FP2	
		Transisi Naik	Transisi Turun	Naik	Turun
Cacahan hanya di T1	Tinggi	Turun	Naik	Tak Mencacah	Tak Mencacah
	Rendah	Naik	Turun	Tak Mencacah	Tak Mencacah
Cacahan di T2	Tinggi	Tak Mencacah	Tak Mencacah	Naik	Turun
	Rendah	Tak Mencacah	Tak Mencacah	Turun	Naik
Cacahan di T1 dan T2	Tinggi	Turun	Naik	Naik	Turun
	Rendah	Naik	Turun	Turun	Naik

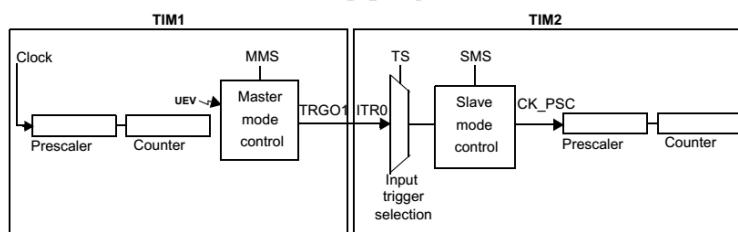
Pembacaan rotary enkoder dengan timer ini hanya akan memperoleh data posisi (putaran) dan arah putaran. Sedangkan untuk memperoleh data kecepatan, percepatan dan perlambatan diperlukan timer lain sebagai dasar pewaktuan yang menghitung nilai cacahan per satuan waktu.



Gambar 3.35 Pembacaan Rotary Enkoder

3.3.2.8 Sinkronisasi Timer

Unit kendali master mempunyai sebuah sinyal keluaran triger (TRGO) yang bisa digunakan sebagai sinyal input triger ke timer lain. Sinyal TRGO juga bisa digunakan sebagai pengatur ADC atau DAC. Ketika dihubungkan dengan timer lain, maka kedua timer akan bekerja secara sinkron, timer pertama bertindak sebagai master sedangkan timer lainnya berrtindak sebagai slave.

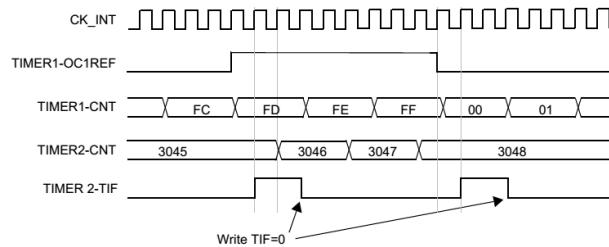


Gambar 3.36 Contoh Timer Master dan Slave

Fungsi pertama yang bisa dilakukan oleh sinkronisasi timer adalah timer master bisa menjadi pra-skalar dari timet slave. Timer master akan mengeluarkan sinyal update secara periodik ke timer slave melalui sinyal TRGO. Timer slave diprogram di mode slave dengan picu internal ITR0. Kendali slave bekerja di clock eksternal mode 1. Sementara timer master bisa mengambil sumber clock internal maupun eksternal. Sinyal TRGO

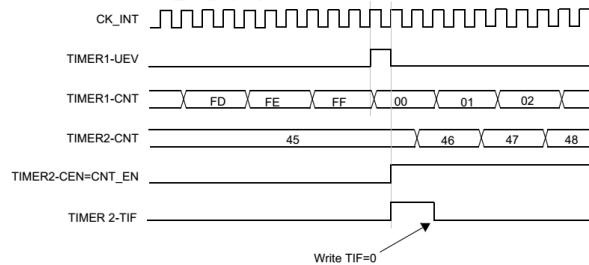
akan terjadi setiap timer master mengalami suatu event (misalnya overflow), dan sinyal ini akan menjadi sumber clock bagi timer slave, sehingga timer master akan bekerja sebagai pra-skalar bagi timer slave.

Fungsi sinkronisasi timer juga bisa digunakan mengaktifkan (*enable*) sebuah timer (slave) oleh timer lain (master). Pada fungsi ini, timer slave diprogram untuk menerima sinyal picu dari timer master dan bekerja di mode gerbang (*gated mode*). Sedangkan timer master diprogram untuk mengirimkan sinyal output compare-nya (OC1REF) sebagai keluaran picu. Pada mode gerbang (akan dijelaskan di sub-bab berikutnya), timer slave hanya akan mulai bekerja (mencacah) jika sinyal OC1REF berlogika tinggi. Dengan mengatur sinyal OC1REF ini, maka timer master bisa mengendalikan aktif tidaknya timer slave.



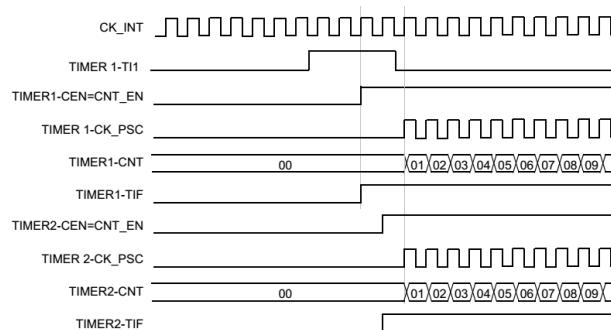
Gambar 3.37 Sinyal OC1REF Timer 1 mengendalikan Timer 2

Selain menggunakan sinyal OC1REF, timer master juga bisa mengaktifkan timer slave menggunakan sinyal UEV (*update event*). Timer master tinggal diprogram untuk mengeluarkan sinyal UEV sebagai output, sedangkan timer slave tetap mendapat input dari timer master.



Gambar 3.38 Sinyal UEV Timer 1 mengendalikan Timer 2

Sinkronisasi timer juga bisa digunakan untuk mensinkronkan 2 buah timer melalui picu eksternal (misal melalui pin TIx). Sebagai contoh Timer 1 dan Timer 2 akan disinkronkan melalui pin TI1. Timer 1 diprogram di mode slave untuk masukan TI1 dan di mode master untuk Timer 2. Timer 2 tetap mendapat sinyal picu dari Timer 1. Pada saat terjadi transisi di pin TI1, kedua timer akan mulai melakukan pencacahan secara sinkron.

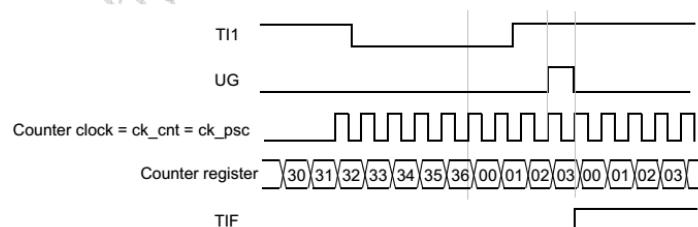


Gambar 3.39 Masukan TI1 Mensinkronkan 2 timer

3.3.2.9 Fungsi Mode Slave

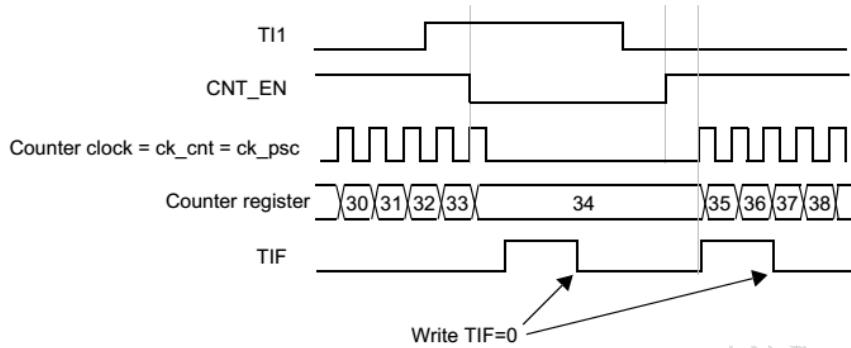
Ada beberapa mode yang bisa dipilih saat sebuah timer bekerja di mode slave dan memperoleh picu eksternal: mode reset (*reset mode*), mode gerbang (*gated mode*) dan mode picu (*trigger mode*).

Pada mode reset, pencacah dan pra-skalaranya bisa diinisialisasi merespon setiap even dari masukan picunya. Misal ketika timer diprogram untuk merespon saat transisi naik di pin TI1, maka pencacah mulai mencacah dari clock internal. Ketika ada transisi naik di TI1, pencacah akan direset dan cacahan kembali ke 0. Contoh aplikasi mode ini adalah pengukuran sinyal PWM.



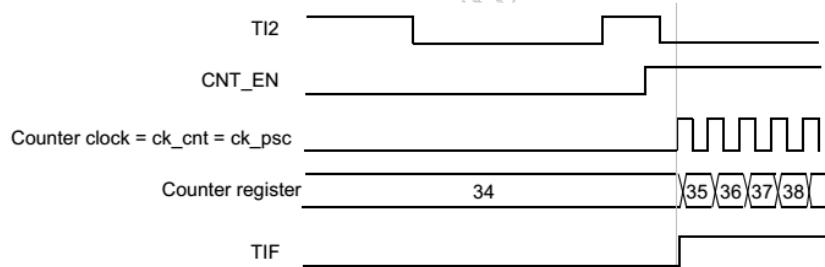
Gambar 3.40 Mode Reset

Mode gerbang mengatur timer hanya akan bekerja (mencacah) pada saat sinyal picunya berada pada level tertentu. Gambar 3.41 menunjukan Timer 1 yang bekerja di mode gerbang dan diatur aktif selama sinyal picu (pin TI1) berada di logika rendah. Selama TI1 berlogika renah Timer 1 akan melakukan caacahan dan berhenti saat TI1 menjadi tinggi.



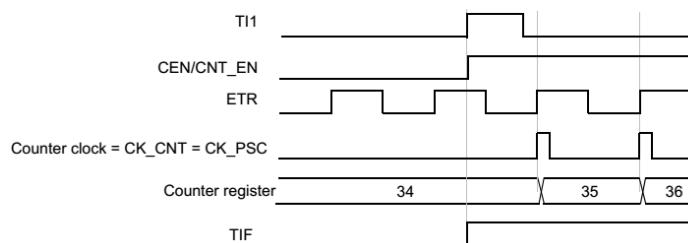
Gambar 3.41 Mode Gerbang

Mode picu mengatur timer merespon atau aktif saat terjadi even di pin masukan, saat terjadi transisi turun atau transisi naik. Gambar 3.42 menunjukan timer di mode picu dan aktif saat transisi naik. Jeda waktu saat terjadi transisi dengan saat mulai pencacahan diakibatkan adanya sinkronisasi pencacahan dengan pin masukan.



Gambar 3.42 Mode Picu

Mode picu jika dikombinasikan dengan clock eksternal mode 2 akan membentuk mode baru. Pada mode ini, timer akan memperoleh clock dari sinyal ETR. Gmbar 3.43 menunjukan timer yang bekerja di mode ini. Timer akan aktif saat ada transisi naik di pin TI dan melakukan cacahan saat transisi naik di ETR.



Gambar 3.43 Mode Picu dan Clock Eksternal Mode 2

3.3.2.10 Mode Debug

Pada saat STM32F207 berada di mode debug (core Cortex-M3 berhenti), timer bisa diatur untuk tetap melakukan cacahan (misal saat pengendalian motor dengan sinyal PWM) atau berhenti melakukan cacahan (misal untuk keperluan watchdog).

3.4 ANALOG TO DIGITAL CONVERTER

STM32F207 dilengkapi dengan 3 buah pengubah digital ke analog (ADC *Analog to Digital Converter*) 12 bit dengan jenis ADC *successive approximation*. Ketiga ADC (ADC1, ADC2, dan ADC3) masing-masing mempunyai 19 saluran yang dimultipleks, sehingga bisa mengukur sinyal analog dari 16 sumber eksternal, khusus ADC1 ditambah 2 sumber internal (sensor suhu dan input vref) dan saluran VBAT. ADC ini bisa bekerja dalam mode sendiri (*single*), terus-menerus (*continuous*), *scan* atau mode tidak terus-menerus (*discontinuous*). Hasil pembacaan ADC disimpan di register 16 bit secara rata kanan (*right-aligned*) atau rata kiri (*left-aligned*). ADC1 adalah ADC master sedangkan ADC2 dan ADC3 merupakan ADC slave. ADC-ADC tersebut bisa bekerja dalam mode dual yang melibatkan 2 ADC atau mode triple yang melibatkan ketiga ADC.

3.4.1 FITUR UTAMA

ADC STM32F207 memiliki beberapa fitur utama:

1. Resolusi bisa diatur menjadi 12, 10, 8 atau 6 bit.
2. Bisa membangkitkan interupsi di akhir konversi, di akhir konversi injected, sat terjadi warchdog analog atau even overrun.
3. Konversi tunggal atau kontinyu

4. Mode scan untuk konversi otomatis dari kanal 0 sampai kanal 'n'.
5. Pengaturan data dengan data bersifat koheren.
6. Waktu pencuplikan (sampling) yang dapat diprogram untuk setiap kanal
7. Pilihan picu eksternal dengan polaritas yang bisa diatur baik untuk konversi reguler atau konversi injected
8. Mode diskontinyu
9. Mode dual/triple untuk yang mempunyai 2 ADC atau lebih
10. Penyimpanan data DMA yang bisa diprogram pada mode dual/triple
11. Waktu tunda (delay)antar konversi yang bisa diprogram pada mode dual/triple interleaved
12. Waktu konversi 0.5 mikro detik pada kecepatan bus APB 60 MHz
13. Membutuhkan catu daya 2.4V sampai 3.6V pada kecepatan tinggi dan turun sampai 1.8V pada kecepatan yang lebih rendah
14. Tegangan masukan VREF- <VIN <VREF+
15. Pembangkitan permintaan ke DMA selama konversi reguler.

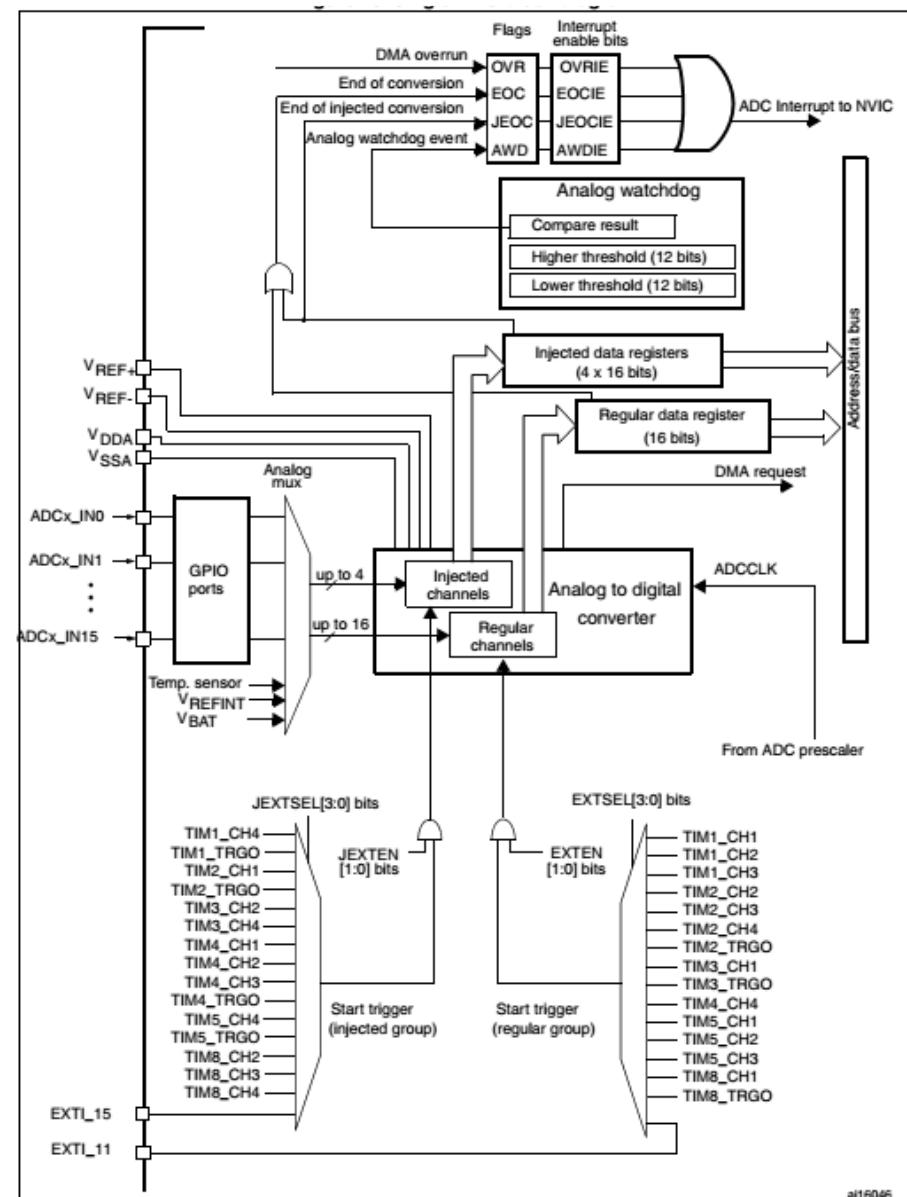
Diagram blok ditunjukan oleh gambar 3.44. Setiap ADC mempunyai kanal regular dan kanal injected. Kanal reguler mempunyai masukan sampai 16 masukan sedangkan kanal injected hanya sampai 4 masukan. Dalam operasinya kanal injected bisa menginterupsi proses konversi yang sedang dilakukan oleh kanal reguler.

Ke-16 masukan eksternal (ADCx_IN0 - ADCx_IN15) dipetakan ke GPIO yang bekerja di fungsi alternatif analog. Ketiga ADC dipetakan ke GPIO yang sama. Semua masukan ADC eksternal maupun internal ini dihubungkan oleh sebuah multiplek analog, karena pada dasarnya ADC hanya bisa melakukan sekali konversi (1 kanal) dalam satu waktu.

Proses konversi bisa dimulai melalui kendali internal ADC dengan menyet bit SWSTART di register ADC_CR2 atau melalui sinyal eksternal, melalui keluaran even sebuah timer atau berasal dari pin interupsi eksternal (EXTI_11 untuk kanal reguler dan EXTI_15 untuk kanal injected).

Sinyal clock untuk proses konversi berasal dari clock APB2. Clock ini akan melalui pra-skalar yang akan membagi frekuensi clock dengan 2, 4, 6 atau 8. Dengan frekuensi APB2 60 MHz maka frekuensi maksimum

clock ADC adalah 30MHz yang akan memberikan waktu konversi 0.5 mikro detik.



Gambar 3.44 Diagram Blok ADC

Data hasil konversi disimpan dalam register 16 bit. Kanal reguler mempunyai 1 buah register data sedangkan register injected mempunyai 4 buah register data. Karena ADC hanya mempunyai resolusi 12 bit, data bisa disimpan secara rata kiri (left alignment) atau rata kanan (right alignment). Tegangan analog hasil konversi dihitung dengan rumus sebagai berikut:

$$V_{KONVERSI} = (VREF * DATA_{ADC}) / 2^{12}$$

ADC juga bisa dijadikan sebagai watchdog analog. Fungsi ini akan membaca level tegangan di input ADC dan membandingkan hasil pembacaanya dengan sebuah nilai ambang batas bawah dan atas. Jika pembacaan ADC berada di luar ambang batas ini, ADC bisa membangkitkan interupsi.

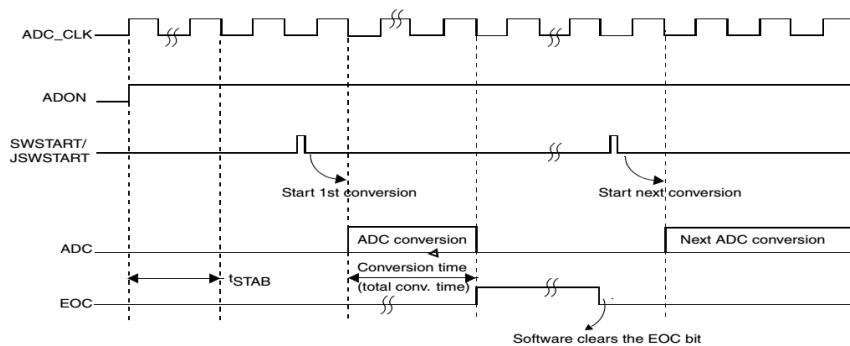
Interupsi ADC bisa dibangkitkan untuk beberapa even sebagai berikut:

1. End of conversion (EOC), setiap akhir konversi kanal reguler
2. DMA Overrun (OVR), terjadi ketika buffer DMA mengalami overflow
3. End of Injected conversion (JEOC), terjadi ketika akhir konversi di kanal injected
4. Analog Watchdog Event (AWD), terjadi saat fungsi watchdog analog diaktifkan dan hasil pembacaan ADC diluar batas ambang bawah dan batas ambang tingginya.

ADC mempunyai pin catu daya terpisah dari sistem catu daya digital. Pin catu daya ini dinamakan dengan VDDA dan VSSA. Dalam aplikasinya pin catu daya ini dihubungkan dengan filter frekuensi tinggi, biasanya sebuah ferrite bead, dengan sistem catu daya digital. Atau bisa juga menggunakan catu daya yang benar-benar terpisah dengan catu daya digital. Untuk tegangan acuan, ADC mempunyai pin VREF+ dan VREF-. Pin VREF- hanya ada di beberapa kemasan dan harus dihubungkan dengan VSSA. Pin VREF+ biasanya dihubungkan dengan VDDA melalui sebuah resistor. Penjelasan terperinci mengenai hal ini akan dibahas di bab selanjutnya.

3.4.2 MODE KERJA

ADC diaktifkan dengan menyet bit ADON di register ADC_CR2 (*Control Register*). Ketika pertama kali bit ini di-set, akan membangunkan ADC dari mode *power down*. Proses konversi diawali setelah bit SWSTART (kanal reguler) atau JSWSTART (kanal injected) di set. Kedua bit ada di register ADC_CR2. ADC memerlukan waktu stabilisasi (t_{STAB}), agar ADC bekerja lebih akurat setelah ADON di set. Setelah proses dimulai dan setelah 15 siklus clock ADC, bit EOC di set dan data ADC disimpan di register data.



Gambar 3.45 Diagram Pewaktuan ADC

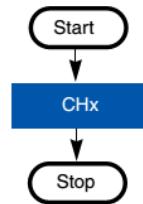
ADC bisa bekerja secara independen maupun secara dual/trippler di mana 2 atau 3 ADC bekerja dalam mode master dan slave. ADC juga bisa melakukan hanya sekali konversi (*single conversion*) atau konversi berulang (*continuous conversion*). Pada mode independen ADC bisa melakukan konversi 1 kanal atau pun multi kanal. Dengan mengurangi resolusi ADC, proses konversi juga bisa dipercepat (*fast conversion*)

3.4.2.1 Mode Independen

Pada mode independen, ADC bisa diprogram untuk beroperasi pada kanal tunggal atau pun kanal jamak dan bisa melakukan proses konversi tunggal atau konversi berulang. Kanal injected juga bisa beroperasi di mode ini.

Mode pertama adalah kanal tunggal dan konversi tunggal. Pada mode ini, ADC melakukan pembacaan di salah satu inputnya, misal ADC1_IN0. Setelah ADC diaktifkan dan proses konversi analog ke digital selesai, data hasil konversi kemudian akan di simpan di register ADC_DR dan tanda EOC aktif dan bisa membangkitkan interupsi jika interupsi

ADC aktif. Setelah itu ADC akan berhenti sampai ada perintah untuk melakukan konversi lagi.



Gambar 3.46 Kanal Tunggal Konversi Tunggal

Mode kedua adalah kanal jamak dengan konversi tunggal. Pada mode ini, ADC diprogram untuk melakukan pembacaan analog terhadap beberapa input analog (sampai 16 kanal) yang telah ditentukan secara berurutan. Setiap kanal bisa diatur waktu samplingnya masing-masing. Setiap selesai melakukan konversi pada sebuah kanal, ADC akan melakukan konversi ke kanal berikutnya sesuai urutan kanal yang telah ditentukan sampai semua kanal tersebut dibaca. Hal ini tentunya bisa menghemat waktu kerja CPU, artinya pada saat proses ini berlangsung, CPU bisa diprogram untuk melakukan fungsi yang lain.



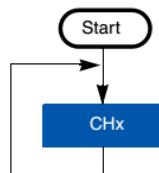
Gambar 3.47 Contoh Urutan Kanal Jamak

Interupsi bisa diprogram setiap setiap selesai proses konversi per kanal atau setelah seluruh kanal selesai. Mengingat register data hanya ada 1, mode ini sebaiknya dilakukan dengan melibatkan DMA, agar setelah selesai konversi data bisa disimpan di buffer DMA (SRAM).



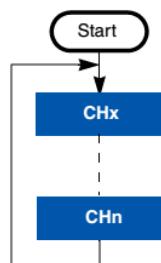
Gambar 3.48 Kanal Jamak Konversi Tunggal

Mode selanjutnya adalah mode kanal tunggal dengan konversi terus-menerus (*single channel continuous conversion*). ADC akan melakukan pembacaan terhadap salah satu kanal yang telah ditentukan dan setelah selesai melakukan konversi, ADC akan secara otomatis melakukan pembacaan lagi tanpa menunggu perintah dari CPU dan juga tanpa melibatkan CPU. Disarankan juga untuk menggunakan DMA dengan mode circular buffer.



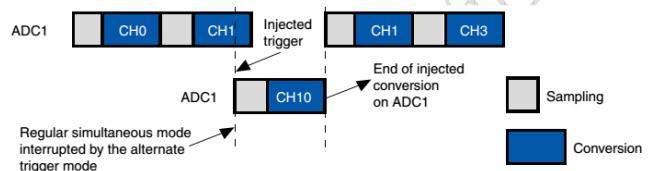
Gambar 3.49 Kanal Tunggal Konversi Terus-Menerus

Mode keempat adalah mode multi kanal dengan konversi yang terus-menerus (*multi channel continuous conversion*). ADC diprogram untuk melakukan pembacaan terhadap beberapa kanal (maksimal 16 kanal) yang telah ditentukan secara berurutan di mana setelah semua kanal dibaca, ADC akan melakukan pembacaan kembali dari kanal urutan pertama dengan tanpa melibatkan CPU. Penggunaan DMA juga disarankan agar data setiap kanal bisa ditransfer ke SRAM.



Gambar 3.50 Kanal Jamak Konversi Terus-Menerus

Keempat mode di atas bisa diterapkan pada kanal reguler mau pun kanal injected. Untuk kanal injected ada mode yang tidak bisa dilakukan oleh kanal reguler. Kanal injected lebih diutamakan dari pada kanal reguler, sehingga kanal injected bisa menginterupsi proses konversi yang sedang dilakukan oleh kanal reguler, artinya ketika reguler sedang melakukan konversi kemudian sebuah sinyal eksternal atau pun software memicu proses konversi kanal injected, maka proses konversi kanal reguler tersebut akan dihentikan terlebih dahulu untuk memproses permintaan konversi kanal injected. Setelah konversi kanal injected selesai, proses konversi kanal reguler akan dilanjutkan kembali.



Gambar 3.51 Kanal Injected Menginterupsi Kanal Reguler

3.4.2.2 Mode Multi ADC

STM32F207 mempunyai 3 ADC (ADC1, ADC2 dan ADC3). Ketiga ADC tersebut bisa bekerja secara independen seperti telah dijelaskan di atas, maupun bekerja pada mode multi ADC sebagai master dan slave, dengan ADC1 bertindak sebagai master. Mode multi ADC bisa bekerja secara dual (2 ADC) mau pun triple (3 ADC). Pada mode multi ADC ini sinyal konversi berasal dari ADC master (ADC1) yang dikirim ke ADC slave secara simultan atau bergantian.

Dalam mode multi ADC, ketika konversi dipicu dari sebuah event eksternal, harus dipastikan hanya ADC master yang diaktifkan mode

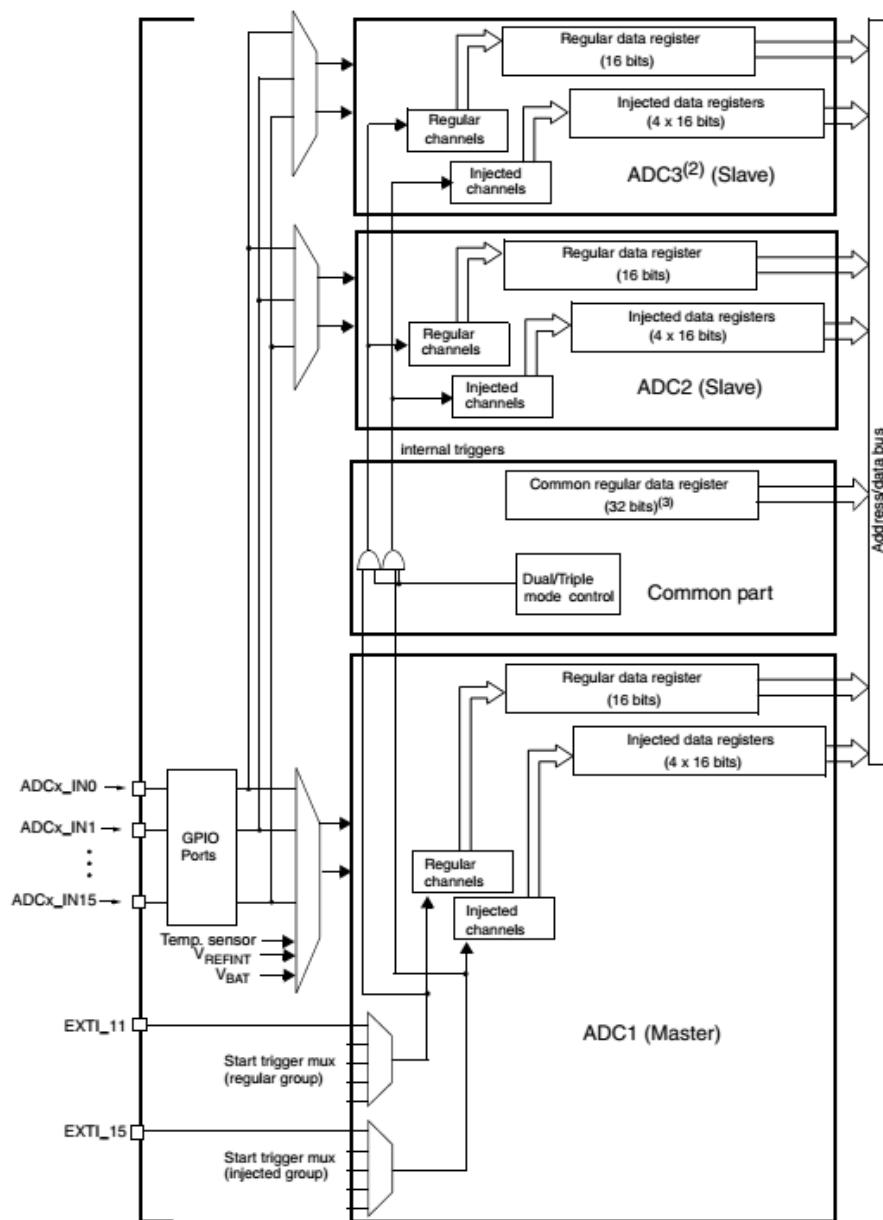
picu eksternalnya. Dan mode picu eksternal ADCslave harus dimatikan untuk menghindari terjadinya proses konversi yang tidak diinginkan.

Ada beberapa mode kerja ADCyang bisa dipilih ketika ADC bekerja secara multi:

1. Mode simultan untuk kanal reguler
2. Mode simultan untuk kanal injected
3. Mode tertinggal (*interleaved*)
4. Mode picu bergantian (*alternate trigger*)

Selain itu ada juga mode gabungan dari mode-mode di atas:

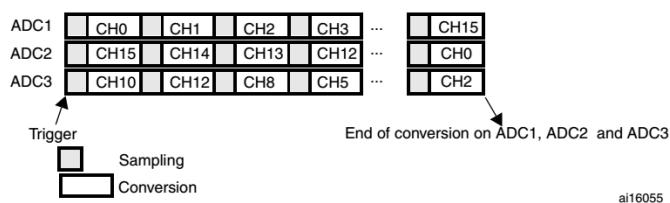
1. Mode simultan gabungan antara kanal reguler dengan kanal injected
2. Mode simultan reguler dan mode picu bergantian



Gambar 3.52 Diagram Blok Multi ADC

Mode simultan bekerja seperti mode kanal jamak tetapi melibatkan dua atau 3 ADC. ADC master dan ADC slave bekerja dengan mode scan terhadap beberapa kanal yang telah ditentukan. Sinyal picu diarahkan ke

ADC1 sebagai ADC master yang secara simultan juga akan memicu ADC slave, ADC2 dan atau ADC3. Dalam mode simultan ini, urutan kanal antara master dan slave tidak boleh sama, artinya dalam setiap konversi kanal antara master dan slave harus berbeda. Proses konversi harus dipastikan berlangsung dalam waktu yang sama atau pastikan rentang waktu sinyal picu lebih lama dari pada waktu terlama antara konversi 2 kanal (pada mode 2 ADC) atau 3 kanal (pada mode 3 ADC).



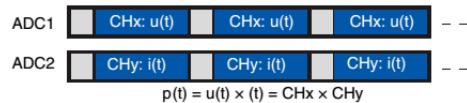
Gambar 3.53 Mode Simultan 3 ADC

Untuk kanal reguler, ketika mode ini aplikasikan, semua kanal injected harus dinonaktifkan agar tidak menginterupsi proses konversi. Tetapi ketika diterapkan ke kanal injected, kanal reguler tetap bisa beroperasi secara independen.

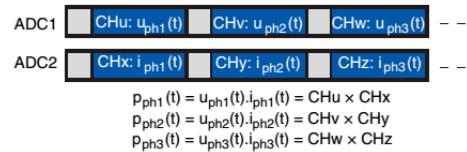
Data hasil konversi simultan akan disimpan di register data gabungan ADC_CDR (Common Data Register). Register ini merupakan register 32 bit yang dibagi menjadi 2 register 16 bit DATA1 (bit ke-0 - bit ke-15) dan DATA2 (bit ke-16 - bit ke-31). Pada mode 2 ADC, DATA1 akan menyimpan data hasil konversi ADC1 dan DATA2 akan menyimpan data hasil konversi ADC2. Sedangkan pada mode 3 ADC, DATA1 akan menyimpan data hasil konversi ADC1, ADC3 dan ADC2 secara bergantian sedangkan DATA2 akan menyimpan data dari ADC2, ADC1 dan ADC3.

Salah satu contoh penggunaan mode simultan adalah ADC dipakai untuk melakukan pengukuran daya listrik. Daya listrik adalah perkalian antara tegangan dan arus. Tegangan dan arus harus dibaca secara bersamaan, sehingga mode simultan sesuai dengan keperluan ini. Misal untuk pengukuran daya menggunakan mode 2 ADC (ADC1 dan ADC2). Untuk pengukuran 1 phase digunakan masing ADC 1 kanal. ADC1 mengukur tegangan sedangkan ADC2 mengukur arus. Untuk pengukuran daya 3 phase digunakan masing-masing 3 kanal, ADC1 mengukur tegangan 3 phase dan ADC2 mengukur arus 3 phase.

Single-phase case:

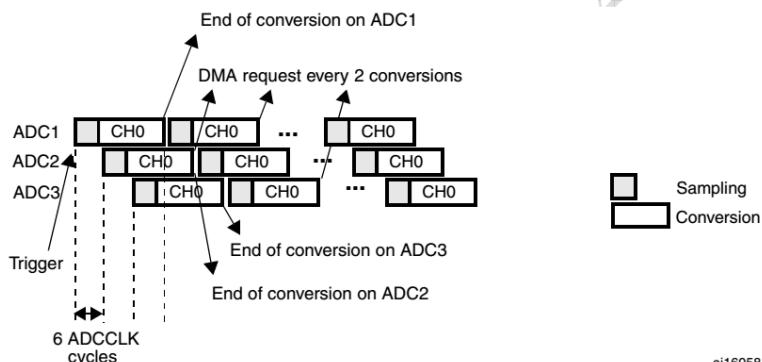


Three-phase case:



Gambar 3.54 Contoh Aplikasi Mode Simultan: Pengukuran Daya Listrik

Mode tertunda (*interleaved*) hanya bisa diterapkan pada kanal reguler. Mode ini hampir sama dengan mode simultan tetapi konversi di ADC slave dimulai beberapa siklus setelah sinyal picu diterima oleh ADC1. Pada mode 3 ADC, ADC3 mulai melakukan konversi beberapa siklus setelah ADC2 melakukan konversi. Artinya waktu mulai konversi ADC slave akan tertunda beberapa siklus setelah ADC master mulai melakukan konversi.



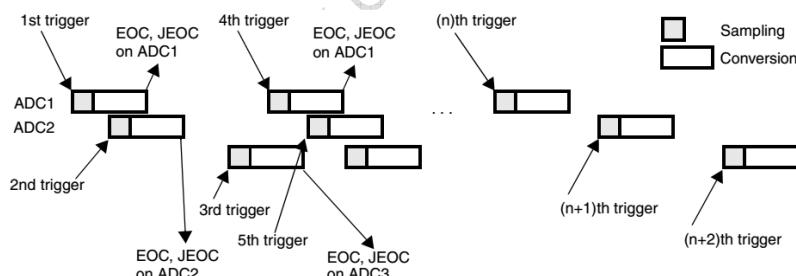
Gambar 3.55 Mode Interleaved 3 ADC

Waktu tunda minimal antara 2 konversi ditentukan oleh bit **DELAY** di register **ADC_CCR**. Tetapi karena ADC hanya bisa melakukan konversi setelah ADC 1 selesai melakukan pencuplikan (sampling) tegangan masukannya, maka waktu tunda antar konversi menjadi 2 kali waktu pencuplikan ditambah 2 siklus clock ADC. Jika **DELAY** = 5 siklus clock dan waktu cuplik 15 siklus clock, maka waktu tunda antar konversi menjadi 17 siklus clock.

Di mode tertunda dengan 2 ADC permintaan DMA akan terjadi setiap selesai proses konversi ADC2. DMA akan menstransfer data hasil konversi ADC2 terlebih dahulu yang tersimpan di register ADC_CDR DATA2 kemudian mentransfer data ADC1 dari register DATA1 ke SRAM. Di mode 3 ADC permintaan DMA akan terjadi ketika telah terdapat data hasil konversi, data hasil konversi yang disimpan di register DATA1 akan ditransfer ke SRAM terlebih dahulu, kemudian diikuti dengan DATA2. Sebagai contoh permintaan pertama DMA akan mentransfer hasil konversi ADC1 terlebih dahulu kemudian hasil konversi ADC2. Setiap kali ada data hasil konversi baru akan disimpan di DATA1, dan DATA1 hasil konversi sebelumnya akan digeser ke DATA2. Urutan permintaan DMA pada mode 3 ADC adalah sebagai berikut:

1. Permintaan pertama: ADC2_DR | ADC1_DR
2. Permintaan kedua: ADC1_DR | ADC3_DR
3. Permintaan ketiga: ADC3_DR | ADC2_DR
4. Pernintaan keempat: ADC2_DR | ADC1_DR

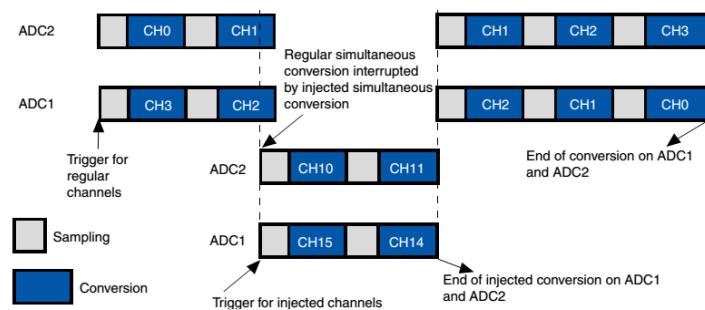
Mode picu bergantian (*alternate trigger*) hanya bisa diterapkan di kanal injected. Dalam mode ini, setiap konversi memerlukan sinyal picu yang dihubungkan ke masukan picu ADC1. Sinyal picu pertama akan memicu proses konversi ADC1, picu kedua memicu ADC2, picu ketiga untuk ADC3. Setelah semua ADC melakukan konversi, sinyal picu berikutnya akan memicu ADC1, pada mode 2 ADC, sinyal picu ketiga akan kembali memicu ADC1 sedangkan pada mode 3 ADC terjadi pada sinyal picu keempat.



Gambar 3.56 Mode Picu Bergantian 3 ADC

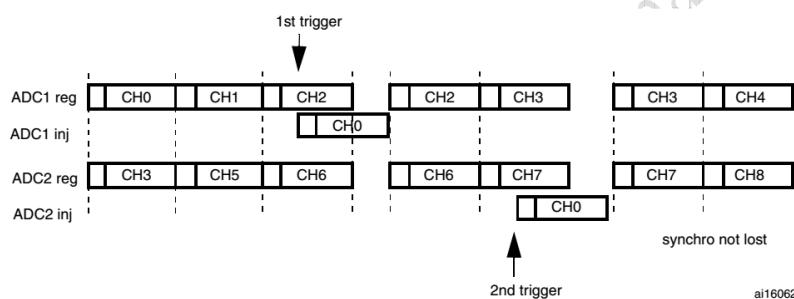
Mode-mode gabungan pada dasarnya memanfaatkan sifat kanal injected yang bisa menginterupsi kanal reguler. Walaupun kanal injected akan

menginterupsi proses konversi kanal reguler, tetapi memungkinkan mode konversi simultan antara kanal reguler dan injected berjalan bersamaan.



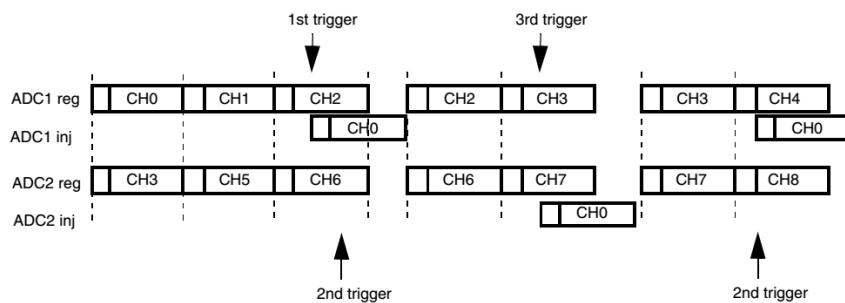
Gambar 3.57 Mode Simultan Gabungan Kanal Reguler dan Injected

Mode gabungan kedua yaitu mode simultan reguler dengan mode picu bergantian (yang hanya untuk kanal injected). Pada mode gabungan ini rentang waktu sinyal picu lebih lama dari pada waktu terlama antara konversi 2 kanal (pada mode 2 ADC) atau 3 kanal (pada mode 3 ADC).



Gambar 3.58 Mode Simultan Reguler dan Mode Picu Bergantian

Ketika sinyal picu terjadi pada saat proses konversi kanal reguler, maka proses konversi kanal reguler akan diinterupsi, sedangkan jika sinyal picu itu datang ketika proses konversi di kanal injected maka sinyal picu itu akan diabaikan.



Gambar 3.59 Sinyal Picu yang Diabaikan

3.4.2.3 Konversi Cepat

Secara normal ADC akan melakukan konversi selama 15 siklus clock ADC. Dengan menurunkan resolusi bit ADC, maka proses konversi bisa dipercepat. Hal ini bisa dilakukan dengan memprogram bit RES[1:0] di register kendali ADC (ADC_CR1). Seperti telah disebutkan bahwa resolusi bisa diatur menjadi 12, 10, 8, atau 6 bit. Kecepatan proses konversi akan menjadi:

1. Resolusi 12 bit: 15 siklus clock ADC
2. Resolusi 10 bit: 13 siklus clock ADC
3. Resolusi 8 bit: 11 siklus clock ADC
4. Resolusi 6 bit: 9 siklus clock ADC

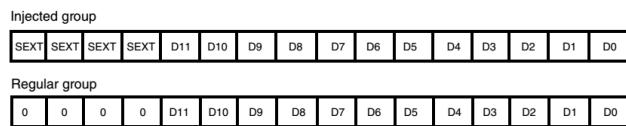
3.4.2.4 Manajemen Data

Data hasil konversi akan disimpan di register ADC_DR untuk operasi ADC tunggal dan register ADC_CDR untuk operasi multi ADC. Software harus segera membaca hasil konversi ini dan memindahkannya ke SRAM sebelum tertimpa data hasil konversi selanjutnya. Ketika proses konversi sangat cepat, misal di mode terus-menerus, atau ketika bekerja di mode multi ADC, disarankan untuk melibatkan kendali DMA sehingga tidak ada data yang hilang.

Ketika DMA diaktifkan untuk ADC, setiap kali selesai konversi permintaan DMA akan dibangkitkan sehingga data dari register ADC_DR bisa ditransfer ke memori SRAM di alamat yang telah ditentukan. Dengan DMA, apabila ada data hilang bisa diketahui dengan melihat bit OVR (*overrun*) di register status ADC (ADC_SR). Jika terjadi hal seperti ini, proses konversi yang sedang berlangsung akan dibatalkan

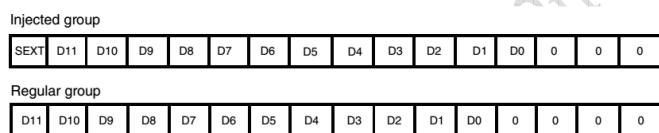
dan juga picu terhadap ADC akan diabaikan. Software harus mereset bit OVR dan menginisialisasi ulang DMA dan ADC. Kecuali di kanal injected yan tidak terperngaruhi oleh error akibat *overrun*.

Seperti telah dijelaskan, STM32F207 mempunyai ADC 12 bit dan hasil konversi akan disimpan di register 16 bit. Dengan mangatur bit ALIGN di register ADC_CR2, data bisa diatur untuk disimpan secara rata kanan (*right alignment*) atau pun rata kiri (*left alignment*).



Gambar 3.60 Data Rata Kanan 12 bit

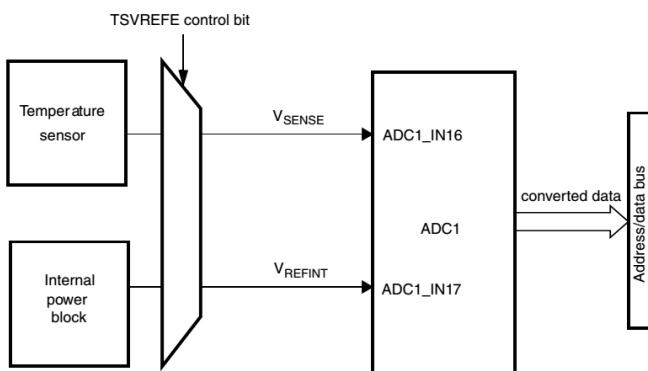
Pada kanal injected, data hasil konversi nilainya akan dikurangi dengan nilai offset yang disimpan di register ADC_JOFRx, sehingga data bisa bernilai negatif. Bit SEXT menunjukan tanda apakah data negatif atau positif.



Gambar 3.61 Data Rata Kiri 12 bit

3.4.2.5 Sensor Suhu

STM32F207 mempunyai sensor suhu yang bisa digunakan untuk mengukur suhu dari prosesor. Sensor suhu ini terhubung dengan ADC secara internal di ADC1_IN16. Sensor suhu ini bisa mengukur suhu dari -40 sampai 125°C dengan toleransi $\pm 1.5^\circ\text{C}$. Pengukuran suhu ini dilakukan dengan men-set bit TSVREFE di register ADC_CCR. Bit ini juga akan mengaktifkan kanal internal ADC1_IN17 yang melakukan pengukuran tegangan di blok catu daya internal. Sensor suhu dan tegangan internal ini hanya bisa dibaca melalui ADC1.



Gambar 3.62 Koneksi Internal Sensor Suhu dan VREFINT

Suhu hasil pengukuran dihitung sebagai berikut:

$$\text{Suhu } (^{\circ}\text{C}) = \{(V_{SENSE} - V_{25}) / \text{Avg_Slope}\} + 25$$

di mana:

V_{SENSE} = data pengukuran ADC

V_{25} = V_{SENSE} pada suhu 25°C

Avg_Slope = kurva rata-rata slope temperatur terhadap V_{SENSE} (dalam mV/°C atau μ V/°C)

Nilai Avg_Slope dan V_{25} bisa dilihat di lembaran data STM32F207.

3.4.2.6 Monitoring VBAT

Pin VBAT bisa dimonitoring dengan membaca ADC_IN18 di ADC1. Oleh karena tegangan di pin VBAT bisa lebih besar dari catu daya di VDDA, sehingga secara otomatis tegangan VBAT ini akan melalui sebuah pembagi tegangan sehingga tegangan yang masuk ke ADC akan menjadi $VBAT/2$.

3.5 DIGITAL TO ANALOG CONVERTER

STM32F207 dilengkapi dengan DAC (*Digital to Analog Converter*) yang mempunyai 2 keluaran yang dipetakan ke 2 pin mikrokontroler. Kedua keluaran ini mempunyai konverter masing-masing. DAC ini mempunyai resolusi 12 bit namun bisa dikonfigurasi menjadi 12 atau 8 bit. Konversi bisa dilakukan pada mode tunggal atau 2 DAC yang berkerja secara

simultan. Untuk konversi ke analog, DAC menggunakan tegangan VREF yang dipakai juga oleh ADC.

3.5.1. FITUR UTAMA

DAC mempunyai beberapa fitur utama:

1. Mempunyai 2 konverter dengan keluaran masing-masing
2. Pembangkitan sinyal noise atau gelombang segitiga
3. Bisa bekerja secara tunggal atau simultan 2 DAC
4. Dihubungkan dengan DMA dengan kemampuan deteksi underrun
5. Sinyak picu eksternal
6. Tegangan acuan ke tegangan VREF

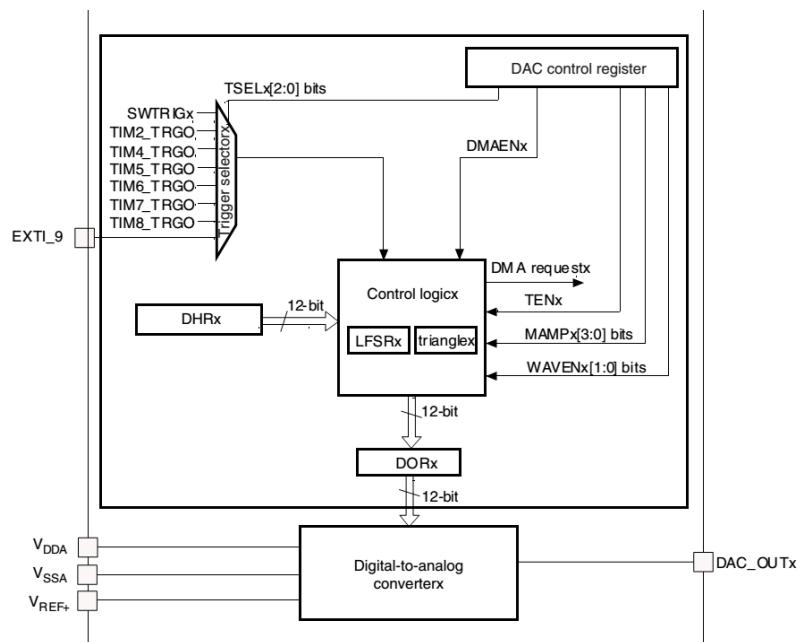
Pada dasarnya DAC, seperti diperlihatkan oleh diagram bloknya, tersusun atas beberapa bagian:

1. Bagian analog atau DAC itu sendiri
2. Bagian kendali
3. Bagian data input dan output
4. Bagian pemicu (trigger)

Bagian analog (DAC) sebagai konverter berfungsi mengubah data digital keluaran bagian kendali (DORx) menjadi sinyal analog. Bagian analog ini mendapatkan catu daya dari pin catu daya yang sama dengan ADC (VDDA dan VSSA). Tegangan acuan untuk proses konversi juga didapat dari tegangan acuan yang sama dengan ADC. Tegangan analog hasil konversi dikeluarkan melalui salah satu pin GPIO yang dipetakan ke DAC, biasanya PA4 dan PA5. Keluaran DAC ini dilengkapi dengan buffer untuk menjaga agar impedansi output tetap rendah.

Bagian kendali berfungsi untuk mengendalikan mode kerja DAC. Bagian ini mendapat sinyal kendali dari register kendali DAC (DAC_CR). Sinyal-sinyal ini berfungsi untuk mengaktifkan sinyal picu (TENx), mengatur amplitudo gelombang yang dibangkitkan (MAMPx) dan menentukan apakah DAC akan menghasilkan gelombang (gelombang noise atau segitiga) atau tidak (WAVENx). Bagian kendali menerima data yang akan diubah ke dalam bentuk analog dari register penyimpan data (*data holding register*) DHRx, yang bisa 12 bit atau 8 bit. Data ini kemudian masuk ke bagian kendali untuk dikeluarkan ke register keluaran (DORx) sebelum

kemudian diubah ke analog. Ketika DMA diaktifkan, bagian kendali juga akan membagikan permintaan DMA.

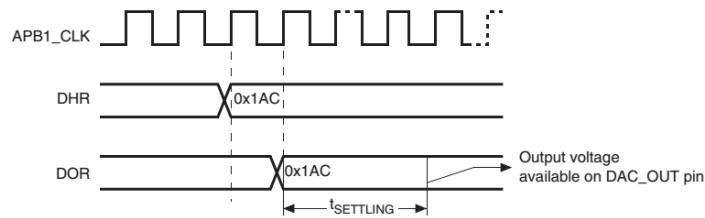


Gambar 3.63 Diagram Blok DAC STM32F207

Untuk pembangkitan gelombang, bagian kendali dilengkapi dengan LFSR (*Linear Feedback Shift Register*) dan TRIANGLEx. LFSR digunakan ketika DAC difungsikan untuk membangkitkan gelombang noise, sedangkan TRIANGLEx digunakan untuk membangkitkan gelombang segitiga.

Bagian pemicu berfungsi untuk memilih sinyal picu untuk memulai proses konversi. Sinyal picu ini bisa terjadi secara software (SWTRIGx) atau berasal dari keluaran timer (TIMx_TRGO) atau bisa juga berasal dari sinyal picu eksternal, melalui pin EXTI_9. Pemilihan sinyal picu dilakukan melalui register DAC_CR.

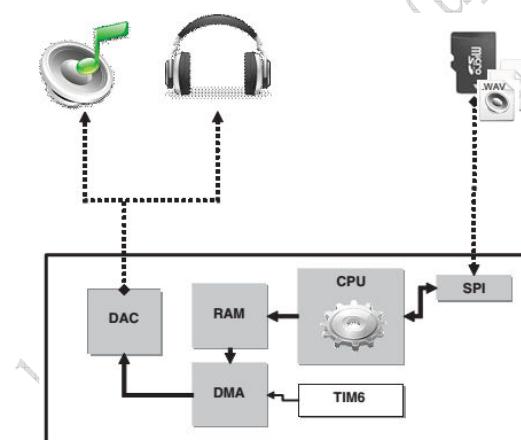
DAC mendeteksi adanya transisi naik di sinyal picunya (timer atau picu eksternal). Data yang tersimpan di register DHR kemudian akan ditransfer ke register DOR, dan proses konversi dimulai. Sinyal picu secara software terjadi ketika bit SWTRG di set, bit ini akan direset secara hardware setelah data di register DHR telah ditransfer ke register DOR.



Gambar 3.64 Diagram Pewaktuan Proses DAC

3.5.2. MODE KERJA

DAC akan mengubah data digital menjadi sinyal analog. Ketika data digital berasal dari musik yang sudah di-digitalkan maka ketika data tersebut dikirim dengan kecepatan yang sama saat musik itu dikonversi ke data digital oleh sebuah ADC, sinyal analog yang dihasilkan akan sama dengan sinyal musik tersebut. DAC STM32F207 bisa digunakan untuk memainkan musik dari file digital, misalnya file wav. Dengan 2 keluaran dan 2 konverter yang dimilikinya DAC ini bisa menghasilkan audio yang stereo, walaupun tentu saja karena bit resolusi dan frekuensi sampling yang terbatas, sinyal audio yang dihasilkan tidak akan setara dengan kualitas CD audio.



Gambar 3.65 Diagram Blok PlayerFile Wav

DAC STM32F207 terhubung dengan bus APB1 yang mempunyai clock maximum 30 MHz. Dengan menggunakan DMA dan piku dari timer, STM32F207 mempunyai kecepatan (*sampling rate*) maksimum 7.5 Mps

(Mega sample per second), sedangkan tegangan keluaran DAC dihitung dengan rumus:

$$V_{DAC} = (\text{DATA}_{DAC} \times VREF) / 4095$$

Tegangan VREF biasanya dihubungkan dengan 3.3V, sehingga tegangan keluaran DAC maksimal adalah 3.3V. Nilai 4095 adalah $2^{12}-1$ (12 bit).

3.5.2.1 Format Data

DAC bisa bekerja dengan resolusi 12 atau 8 bit, dan data bisa diatur untuk rata kanan maupun rata kiri. Pada mode 8 bit data hanya bisa diatur untuk rata kanan. Setiap resolusi dan pengaturan data mempunyai register data (DHRx) yang berbeda. Begitu juga untuk mode tunggal dan mode 2 DAC. Sehingga software harus memperhatikan format data ini agar menulis data ke register data yang tepat. Untuk mode 1 DAC, data masing-masing kanal juga terpisah, sedangkan untuk mode dual, data masing-masing kanal tersimpan dalam 1 register. Berikut register untuk masing-masing mode:

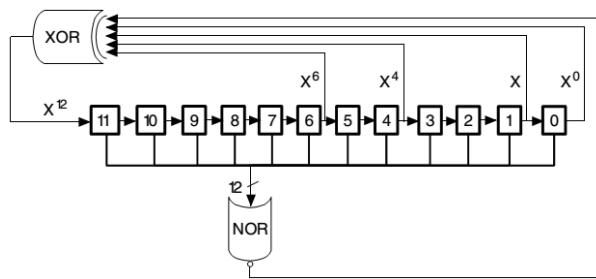
1. DAC_DHR12R1 dan DAC_DHR12R2, register data untuk resolusi 12 bit rata kanan, kanal 1 dan kanal2
2. DAC_DHR12L1 dan DAC_DHR12L2, register data untuk resolusi 12 bit rata kiri
3. DAC_DHR8R1 dan DAC_DHR8R2, register data resolusi 8 bit rata kanan
4. DAC_DHR12RD dan DAC_DHR12LD, register untuk resolusi 12 bit, dual DAC, rata kanan dan rata kiri.
5. DAC_DHR8RD, register data untuk resolusi 8 bit, dual DAC rata kanan

Sedangkan untuk register data output (DOR), DAC hanya punya 2 register yaitu DAC_DOR1 dan DAC_DOR2.

3.5.2.2 Pembangkitan Sinyal Noise

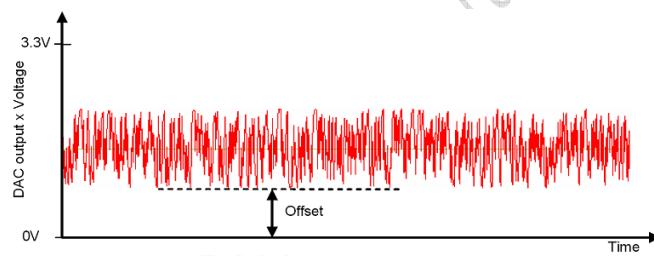
LFSR pada dasarnya adalah sebuah pembangkit kode acak (*pseudo random code generator*) seperti ditunjukkan oleh gambar 3.66. Tergantung dari nilai awal dari register geser, pembangkit ini bisa menghasilkan bilangan berurut sampai 2 sebelum kemudian mengulang dari awal. Dengan adanya LFSR ini DAC bisa digunakan untuk menghasilkan sinyal

noise. Sinyal noise yang dihasilkan terdistribusi secara spektrum secara merata, sehingga bisa dianggap sebagai noise putih (white noise).



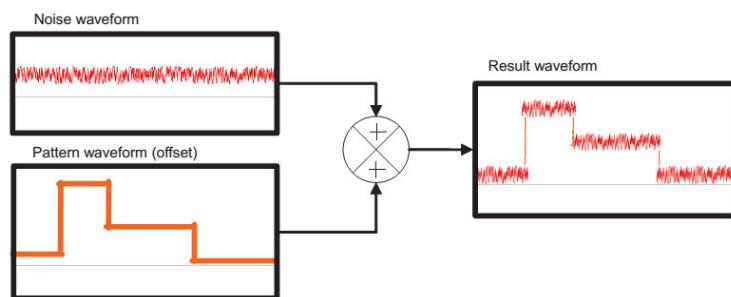
Gambar 3.66 Algoritma Perhitungan Register LFSR

Pembangkitan sinyal noise diaktifkan dengan men-set bit-bit WAVEEx[1:0] register DAC_CR "01". LFSR akan diisi dengan nilai awal 0xAAA dan akan diperbarui setelah 3 siklus clock APB1 setelah sinyal picu diterima mengikuti algoritma yang ditunjukan oleh gambar 3.66. Nilai LFSR ini kemudian ditambahkan dengan register DAC_DHRx sebelum kemudian dikeluarkan melalui register DAC_DORx. Nilai awal register DAC_DHRx ini akan menghasilkan tegangan offset DC di sinyal noise yang dihasilkan.



Gambar 3.67 Sinyal Noise Yang Dihasilkan

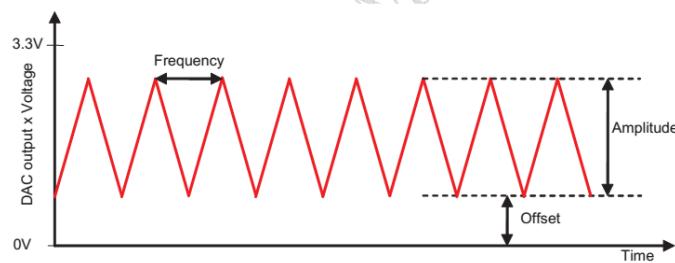
Dengan mengatur data offset (register DAC_DHRx) dengan pola tertentu melalui tabel data, bisa menghasilkan gelombang noise dengan pola tertentu yang berguna untuk menghasilkan sinyal sebuah instrumen musik, misalnya cymbal yang memang gelombang audio yang dihasilkannya banyak mengandung frekuensi tinggi yang mirip dengan noise.



Gambar 3.68 Sinyal Noise dengan Offset yang Diatur

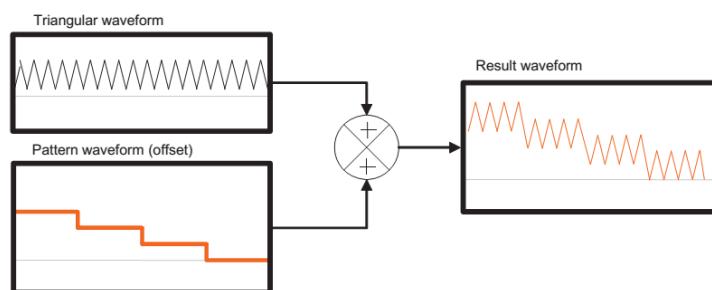
3.5.2.3 Pembangkitan Gelombang Segitiga

Pembangkitan gelombang segitiga dilakukan dengan memprogram bit-bit WAVE[1:0] menjadi "10". Untuk pembangkitan sinyal ini, STM32F207 dilengkapi dengan sebuah pencacah segitiga. Pencacah ini akan diperbaharui setelah 3 siklus clock APB1 setelah sinyal picu diterima. Data pencacah ini akan ditambahkan dengan register DAC_DHR. Nilai pencacah akan terus dinaikkan selama belum mencapai nilai amplida maksimum yang disimpan di register MAMPx. Ketika sudah mencapai maksimum, nilai pencacah akan diturunkan sampai mencapai nol untuk kemudian dinaikkan lagi.



Gambar 3.69 Gelombang Segitiga yang Dihasilkan

Seperti halnya sinyal noise, gelombang segitiga juga akan mempunyai tegangan offset yanh ditentukan oleh register DAC_DHR. Dengan mengatur nilai offset ini dengan pola tertentu bisa dihasilkan sinyal segitiga dengan pola tertentu juga.



Gambar 3.70 Gelombang Segitiga dengan Offset yang Diatur

3.5.2.4 Mode Dual DAC

Kedua kanal DAC bisa diprogram untuk bekerja di mode dual. Pada mode ini, kedua kanal akan menggunakan register data yang memang dikhususkan untuk mode dual yaitu register DAC_DHR12RD, DAC_DHR12LD, dan register DAC_DHR8RD. Mode dual ini bisa lebih mengefisienkan pemakaian bus data karena kedua kanal akan dikendalikan dalam waktu yang sama. Walaupun kedua kanal akan menggunakan register data yang sama, tetapi kendali yang lain seperti sinyal picu atau pengaturan amplitudo bisa diatur secara independen.

Setidaknya ada 11 mode yang bisa dipilih di mode dual ini, 5 mode dengan sinyal picu independen, 5 mode dengan sinyal picu simultan dan 1 mode menggunakan mode picu software bersama mengingat hanya ada 1 picu secara software. Berikut mode-mode dual DAC:

1. Mode picu independen, tanpa membangkitkan gelombang. Kedua DAC akan menghasilkan keluaran berdasarkan register data dengan sinyal picu yang berbeda.
2. Mode picu independen dengan pembangkitan sinyal noise yang sama. Kedua DAC mempunyai sinyal picu masing-masing, tetapi register MAMP kedua DAC diisi data yang sama.
3. Mode picu independen dengan pembangkitan noise yang berbeda. Kedua DAC mempunyai sinyal picu dan isi MAMP yang berbeda.
4. Mode picu independen dengan pembangkitan gelombang segitiga yang sama. Kedua DAC akan mengeluarkan gelombang segitiga dengan amplitudo yang sama (nilai MAMP sama), tetapi sumber sinyal picu yang berbeda.

5. Mode independen dengan pembangkitan gelombang segitiga yang berbeda. Kedua DAC memiliki picu dan amplitudo gelombang segitiga yang berbeda.
6. Mode dipicu bersamaan melalui software.
7. Mode picu bersamaan tanpa pembangkitan gelombang. DAC dipicu dari sumber yang sama dan dari register data.
8. Mode picu bersamaan pembangkitan sinyal noise yang sama. Sumber picu berasal dari sumber yang sama dan nilai MAMP kedua DAC dibuat sama.
9. Mode picu bersama pembangkitan sinyal noise yang berbeda. Sumber picu berasal dari sumber yang sama dan nilai MAMP kedua DAC dibuat berbeda.
10. Mode picu bersama pembangkitan gelombang segitiga yang berbeda. Sinyal picu berasal dari sumber yang sama, amplitudo gelombang kedua kanal dibuat berbeda.
11. Mode picu bersama pembangkitan gelombang segitiga yang sama. Sinyal picu berasal dari sumber yang sama dan amplitudo gelombang dibuat sama.

3.6 KOMUNIKASI SERIAL UART/USART

STM32F207 dilengkapi sampai 6 port serial, dari 6 port serial ini 4 port merupakan USART (Universal Synchronous/Asynchronous Receiver/Transmitter) dan 2 merupakan UART (Universal Asynchronous Receiver/Transmitter).

USART lebih kompleks dari pada UART. Pada UART, pengirim dan penerima harus mengetahui kecepatan data (baud rate), karena data dibaca melalui pendekripsi bit stop, kemudian dari baud rate penerima mencuplik aliran bit data tersebut. Kalau baud rate pengirim dan penerima tidak sama, data dipastikan akan error. Pada USART, data bisa dikirim secara sinkron, artinya pengirim akan mengirimkan sinyal clock sinkronisasi agar penerima bisa menerima data dengan benar tanpa perlu mengetahui kecepatan data. Dengan adanya sinyal clock sinkronisasi ini, kecepatan data USART bisa lebih tinggi dari pada kecepatan data UART. Kecepatan data UART maksimum sampai 4 Mbps sedangkan USART

bisa mencapai 7.5 Mbps. Tentu saja USART memerlukan pin tambahan yaitu pin untuk mengirimkan sinyal clock.

Contoh aplikasi yang menggunakan USART misalnya komunikasi Infra merah IrDA (Infra red Data Association), Smart card atau jaringan LIN (Local Interconnect Network).

3.6.1 FITUR UTAMA

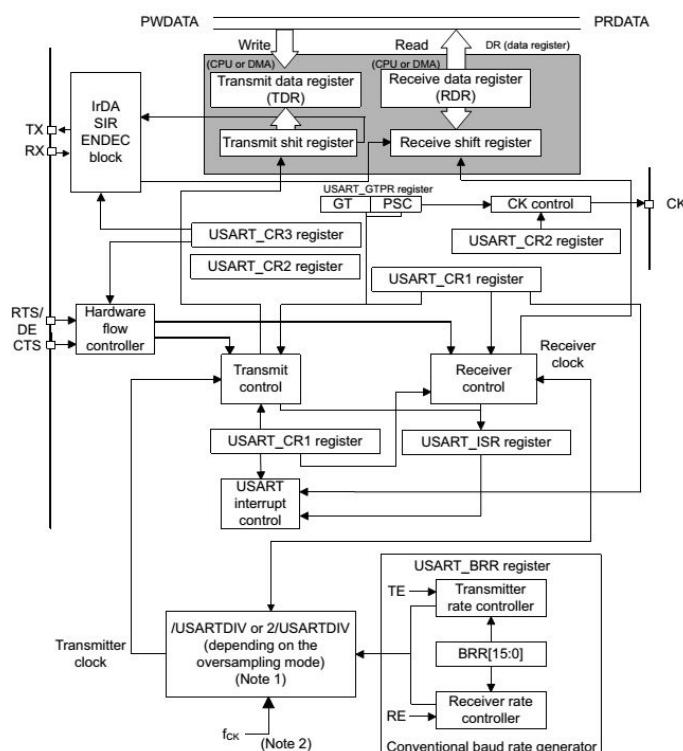
USART/UART mempunyai beberapa fitur utama:

1. Komunikasi asinkron dupleks penuh dan mendukung komunikasi modem penuh dengan kendali aliran (flow control) melalui pin RTS dan CTS.
2. Format NRZ (Non Return to Zero) standar (mark/space)
3. Oversampling yang bisa diatur (16 atau 8) untuk menyesuaikan kecepatan data dan toleransi clock.
4. Kecepatan yang dapat diatur sampai 7.5 Mbps pada clock APB 60 MHz.
5. Panjang data dan bit stop yang bisa diatur. Panjang data bisa dipilih 8 atau 9 bit, sedangkan bit stop bisa dipilih 1 atau 2 bit.
6. Kendali bit paritas (ganjil atau genap) pada penerimaan data.
7. Mendukung jaringan LIN
8. Mendukung komunikasi IrDA dengan durasi 3/16 bit mode normal.
9. Mendukung antarmuka Smartcard dengan standar ISO7816-3.
10. Mendukung komunikasi setengah duplek wire tunggal (single wire)
11. Bisa diatur untuk mengirim dan terima, hanya pengirim atau hanya penerima
12. Bisa dikendalikan melalui DMA dengan buffer yang bisa diatur.
13. Mendukung tanda (flag) buffer penerima penuh, buffer pengiriman kosong, dan tanda akhir pengiriman.
14. Mendukung tanda (flag) pendekripsi error: error overrun, error karena noise, error di frame data dan error karena bit paritas.
15. Mempunyai 10 sumber interupsi: perubahan di pin CTS, deteksi kondisi break di komunikasi LIN, register data pengiriman kosong, transmisi komplit, register data penerima penuh, diterimanya saluran

- idle, saat error overrun, terdeteksi noise, error di frame data, dan saat error di bit paritas.
16. Mendukung komunikasi multiprosesor dengan fungsi mute saat alamat yang diterima tidak sesuai dan aktif lagi saat alamat sesuai.

3.6.2 MODE KERJA

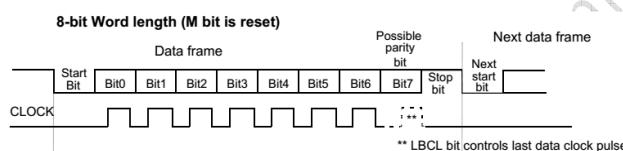
Sebuah perangkat komunikasi serial UART biasanya akan terdiri atas register data untuk pengiriman dan penerimaan data, register kendali untuk mengatur fungsi dan mode kerja UART, register status yang akan memberikan informasi status port serial dan pembangkit baud rate. Untuk koneksi atau antarmukanya akan ada pin RX (penerima) dan pin TX (pengirim). Jika USART/UART tersebut mempunyai kendali aliran (flow control) maka akan ditambah dengan pin CTS atau RTS. Khusus unutuk mode USART (sinkron) akan ada pin untuk mengirimkan clock sinkronisasi (CK).



Gambar 3.71 Diagram Blok USART

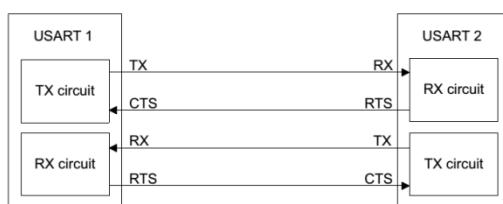
Semua USART/UART STM32F207 terhubung dengan bus APB. USART1 dan USART6 terhubung dengan bus APB2, sedangkan USART2, USART3, USART4 dan USART5 terhubung dengan bus APB1. Clock yang diperlukan akan diambil dari clock APB. Dalam perangkat serial, menulis ke register data pengiriman artinya akan mengirimkan byte data tersebut secara serial (bit demi bit) melalui pin GPIO yang fungsi alternatifnya diprogram ke port serial (sebagai pin TxD). Begitu juga membaca data penerimaan akan membaca data yang telah diterima melalui pin RxD.

Pengiriman dan penerimaan data diawali dengan bit mulai (start bit) dan diakhiri dengan bit berhenti (stop bit). Bit berhenti bisa diatur untuk memiliki panjang bit 0.5, 1, 1.5 atau 2 bit. Sedangkan panjang data bisa diatur menjadi 8 atau 9 bit dan bisa juga menggunakan bit paritas, genap atau ganjil. Jika digunakan bit paritas, panjang data digunakan 9 bit. Pada mode sinkron bit-bit dikirim atau diterima saat ada transisi di sinyal CK. Tidak ada clock sinkronisasi saat pengirimam atau penerimaan bit mulai dan bit berhenti.



Gambar 3.72 Frame Data Serial

Jika kendali aliran diaktifkan melalui pin CTS dan pin RTS maka pengiriman dan penerimaan data akan membaca kondisi kedua pin ini. Pin CTS (*Clear To Send*) akan menjadi input. Saat pin ini terbaca logika rendah menandakan bahwa pihak penerima (perangkat luar misal sebuah modem) siap untuk menerima data. Pin RTS (*Request To Send*) akan menjadi output yang akan menandakan USART STM32F207 siap menerima data. Koneksi pin CTS dan RTS dengan perangkat luar adalah koneksi silang (cross) sama dengan koneksi RX dan TX.



Gambar 3.73 Komunikasi UART dengan Kendali Alir RTS dan CTS

Pengirim dan penerima akan mempunyai baud rate yang sama. Clock untuk membangkitkan baud rate berasal dari clock bus APB. Baud rate dibangkitkan dengan membagi clock tersebut agar sesuai dengan baud rate yang diinginkan, dengan persamaan sebagai berikut:

$$\text{Baud Rate} = \frac{f_{CK}}{8 \times (2 - \text{OVER8}) \times \text{USARTDIV}}$$

Sedangkan baud rate untuk mode Smartcard, LIN dan IrDA baud rate ditentukan dengan persamaan:

$$\text{Baud Rate} = \frac{f_{CK}}{16 \times \text{USARTDIV}}$$

OVER8 adalah metode oversampling (8 atau 16) dan bit ke-15 register kendali USART_CR1. Bernilai 1 jika digunakan oversampling 8 dan bernilai 0 jika oversampling 16. Pada mode Smartcard, LIN dan IrDA hanya bisa digunakan oversampling 16. Sedangkan USARTDIV merupakan nilai pembagi dan disimpan di register USART_BRR. USARTDIV ini terdiri atas 12 bit mantissa (DIV_Mantissa) dan 4 bit pecahan (DIV_Fraction). Ketika OVER8=1, bit DIV_Fraction hanya diperhitungkan 3 bit dan bit ke-3nya harus dijaga 0.

Simbol f_{CK} merupakan frekuensi clock bus APB, 60 MHz untuk APB2 (PCLK2) dan 30 MHz untuk APB1 (PCLK1). Nilai oversampling dipilih dengan pertimbangan kecepatan atau ketepatan (akurasi) baud rate. Akurasi ini dibutuhkan terutama saat menerima data. Oversampling 8 (OVER8=1) dipilih jika diinginkan kecepatan atau baud rate tinggi. Frekuensi maksimal f_{CK} adalah PCLK/8. Pada kondisi ini nilai toleransi penerima pada pergeseran baud rate menjadi rendah. Artinya tingkat akurasi baud rate harus tinggi. Sedangkan oversampling 16 digunakan untuk meningkatkan tingkat toleransi penerima pada pergeseran atau error di baud rate. Namun kecepatan data akan dibatasi karena f_{CK} akan mempunyai nilai maksimal PCLK/16.

Dari persamaan baud rate di atas, nilai USARTDIV bisa dihitung dengan metode sebagai berikut. Misal USART6 digunakan untuk mengendalikan modem GSM dengan baud rate 115200 bps. Seperti disebutkan di atas, USART6 ini terhubung dengan bus APB2 yang mempunyai frekuensi clock 60 MHz. Misal digunakan oversampling 8 (OVER8=1). Maka:

$$\text{USARTDIV} = \frac{f_{CK}}{8 \times (2 - \text{OVER8}) \times \text{Baud Rate}}$$

Akan didapat:

$$\text{USARTDIV} = \frac{60000000}{8 \times (2-1) \times 11520}$$

$$\text{USARTDIV} = 65.10417$$

DIV_Mantissa akan bernilai = 65 atau 0x41. Dan DIV_Fraction akan menjadi $8 \times 0.10417 = 0.83336$ (dikali 8 karena digunakan oversampling 8). Nilai bilangan bulat yang paling mendekati adalah 1 atau 0x01. Sehingga nilai USARTDIV atau register USART_BRR akan menjadi 0x411. Dengan nilai BRR 0x411, nilai USARTDIV sebenarnya adalah 65.1 dan baud rate yang dihasilkan akan menjadi 115207.4 bps.

Dengan persamaan di atas, STM32F207 mempunyai fleksibilitas untuk mengatur baud rate sesuai baud rate standar yang biasa dipakai di komunikasi serial. Referensi Manual STM32F207 memuat tabel mengenai baud rate di berbagai frekuensi PCLK dan nilai oversampling.

3.6.2.1 Komunikasi Muli-Prosesor

Beberapa STM32F207 bisa terhubung melalui UART yang membentuk sebuah jaringan. Salah satu STM32F207 berfungsi sebagai master di mana pin TXnya terhubung ke semua pin RXSTM32F207 lain yang berfungsi sebagai slave. Dan pin TX semua slave akan terhubung ke pin RX prosesor master. Semua pinTX ini secara logika akan di-AND-kan sebelum masuk ke pin RX prosesor master.

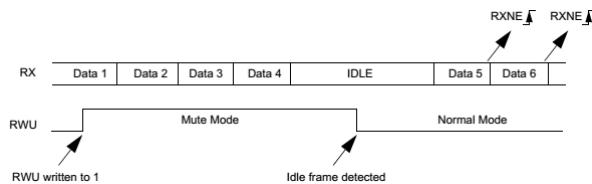
Dalam komunikasi multi-prosesor, biasanya master yang mempunyai inisiatif komunikasi dan hanya satu slave yang dimaksud yang akan menanggapi pesan data yang dikirimkan oleh master. Hal ini dilakukan dengan menggunakan pengalaman yang unik untuk setiap slave. Slave yang mempunyai alamat yang tidak sesuai akan berada di kondisi mute. Kondisi mute ini akan menyebabkan prosesor slave berada di kondisi:

1. Tidak akan ada bit status yang di-set
2. Interupsi serial karena menerima data dilarang
3. Bit RWU (Receiver Wake Up) di register USART_CR1 akan di-set sebagai tanda bahwa penerima serial dalam kondisi mute. Bit RWU ini bisa di-set secara hardware maupun software.

Port serial bisa keluar dari kondisi mute ini bisa dengan 2 cara:

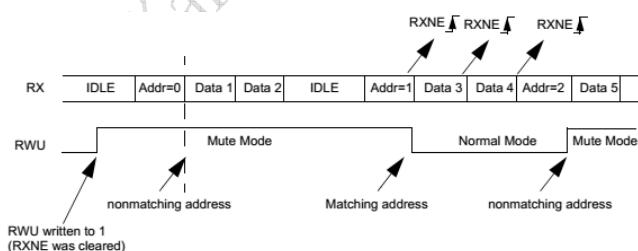
1. Deteksi frame idle, saat bit WAKE dalam kondisi reset
2. Deteksi frame alamat saat bit WAKE di-set.

Setelah bit RWU di-set, USART akan masuk ke mode mute. Ketika frame *idle* terdeteksi, USART akan aktif kembali dan bisa menerima data-data dari pengirim. Untuk kembali ke mode mute, bit RWU harus di-set lagi secara software.



Gambar 3.74 Deteksi Idle

Pada mode pendekripsi alamat dilakukan dengan mendekripsi alamat yang dikirimkan oleh prosesor master. Byte alamat disyaratkan agar semua bit MSBnya (bit 4-bit 7) bernilai satu dan alamat sebenarnya ditentukan oleh 4 bit LSB (bit 0-bit3). Semua byte yang bit-bit MSBnya bernilai 1 akan diperlakukan sebagai byte alamat, selain itu akan dianggap sebagai data. Dengan demikian prosesor slave akan mempunyai alamat dari 0xF0 sampai 0xFF. Alamat ini kemudian akan dibandingkan dengan alamat prosesor yang disimpan di register USART_CR2 (bit ADD). Ketika alamat yang diterima sama, prosesor akan keluar dari mode mute dan akan menerima data-data yang dikirim oleh mastet. Jika kembali menerima alamat yang tidak sesuai, prosesor akan kembali ke mode mute.



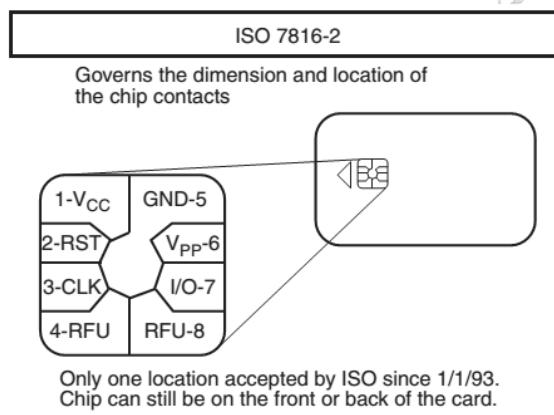
Gambar 3.75 Deteksi Alamat

3.6.2.2 Antarmuka SmartCard

Smartcard atau CCID (*Circuit Card Interface Device*) banyak digunakan untuk kartu kredit, kartu telepon seluler (SIM Card), kartu ATM atau kartu TV berlangganan. Smartcard merupakan sistem berbasis mikrokontroler juga berserta memori yang dicetak dalam bentuk kartu. Smartcard ini mempunyai antarmuka dan protokol komunikasi salah satunya standar ISO-7816-3/4.

ISO-7816 menyediakan standar Smartcard atau disebut juga ICC (Integrated Circuit Card) tentang protokol, keamanan dan antarmuka untuk pembaca smart card. Standar ini meliputi:

1. Bagian 1: Karakteristik Fisik
2. Bagian 2: Ukuran dan posisi kontak
3. Bagian 3: Antarmuka elektrik dan protokol transmisi
4. Bagian 3: Perubahan 2 - Perubahan tentang protokol
5. Bagian 4: Organisasi, keamanan dan perintah untuk komunikasi
6. Bagian 5: Registrasi untuk penyedia aplikasi



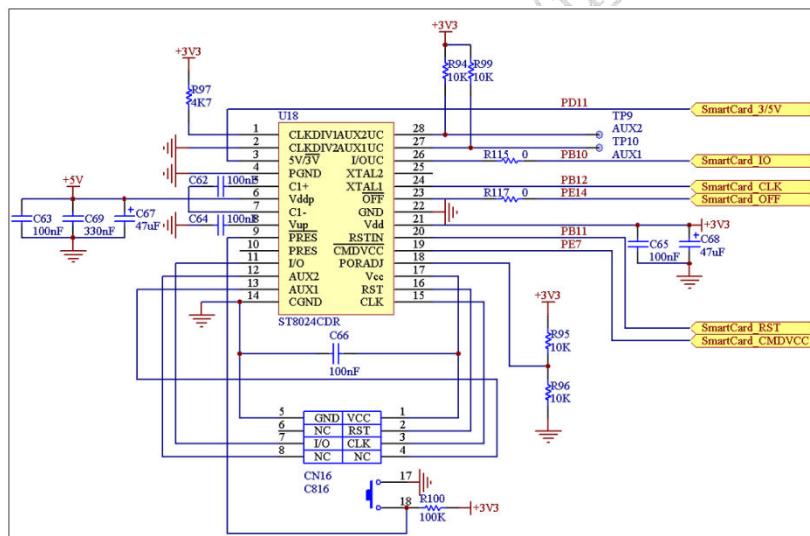
Gambar 3.76 Definisi Pin Smartcard

Smartcard mempunyai antarmuka seperti ditunjukkan oleh gambar 3.76. VCC dan GND merupakan pin catu daya. Smartcard bisa beroperasi pada tegangan 5V atau 3.3V. Pin VPP merupakan pin untuk memberikan tegangan untuk memprogram EEPROM internal smartcard, tegangannya biasanya lebih besar dari VCC. Tetapi untuk smartcard terbaru pin ini tidak digunakan. Pin RST digunakan untuk

mereset smart card. Sedangkan pin I/O digunakan untuk transmisi data, mengirim dan menerima data dalam 1 pin, sehingga mode komunikasinya adalah 2 arah bergantian (*half duplex*). Data dikirim secara asinkron. Pin CLK digunakan untuk memberikan clock untuk mikroprosesor yang ada di smart card. Pin RFU (*Reserved for Future Use*) digunakan untuk pengembangan smartcard selanjutnya.

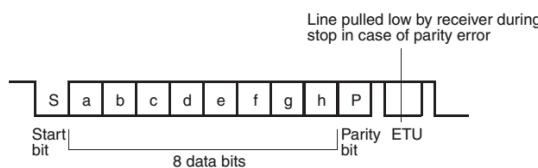
Untuk berkomunikasi dengan smartcard digunakan USART (USART1, USART2, USART3 atau USART6). USART ini akan menyediakan sinyal data (I/O) dan clock sinkronisasi (CLK). Sedangkan untuk pin RST digunakan pin GPIO. Begitu juga untuk VCC digunakan pin GPIO. Namun karena SuTM32F207 pin GPIOnya berada di level tegangan 3.3V, maka STM32F207 hanya bisa menyediakan tegangan 3.3V untuk smartcard.

Selain koneksi langsung, antarmuka dengan smartcard bisa juga melalui sebuah IC driver smart card, misalnya ST8024. Dengan menggunakan IC ini, tegangan smartcard bisa dipilih untuk menggunakan 5V atau 3.3V. Selain itu dengan menggunakan IC ini, bisa digunakan untuk mendeteksi tercabutnya smartcard saat transaksi serta fitur-fitur lain seperti tertulis di lembaran data ST8024.



Gambar 3.77 Contoh Rangkaian Menggunakan IC Driver Smartcard

Standar ISO-7816 untuk smartcard mensyaratkan bahwa USART harus diatur pada mode asinkron dengan bit paritas (misal paritas ganjil) dengan bit stop yang bisa diatur di 0.5 atau 1.5 bit. ISO-7816 juga mendefinisikan waktu bit untuk protokol asinkron sebagai ETU (*Elementary Time Unit*). ETU ini berhubungan erat dengan frekuensi clock yang dikirimkan ke smartcard. Frekuensi clock ini digunakan menangani data serial asinkron oleh smartcard dan juga oleh CPU internal dari smartcard tersebut.



Gambar 3.78 Format Data Serial Smartcard

Saat difungsikan pada mode smartcard, pin TXD akan menjadi pin dua arah (bidirectional), bit-bit data akan dikirim dan diterima melalui pin ini karena smartcard merupakan mode komunikasi kabel tunggal. Oleh karena itu, pin ini harus dikonfigurasi pada mode open drain. Dan USART bisa menyediakan sinyal clock untuk smartcard melalui pin CLK. Sinyal clock ini berasal dari clock periperal dengan praskalar 5 bit. Frekuensi sinyal clock bisa diatur dari $f_{CX}/2$ sampai $f_{CX}/62$.

Keterangan lebih lengkap dari smartcard ini bisa dibaca dicatatkan aplikasi yang dikeluarkan dan dipublikasikan di halaman web ST Microelectronics yaitu *AN4800 Application Note: Smartcard Interface Based on STM32Cube Firmware*.

3.6.2.3 Mode LIN

LIN (*Local Interconnect Network*) merupakan jaringan komunikasi serial yang dipakai otomotif (kendaraan) yang digunakan untuk mengendalikan beberapa komponen di kendaraan misalnya sensor lampu, pintu, dan mungkin beberapa bagian mesin. Jaringan LIN ini dimaksudkan sebagai jaringan pendukung dari jaringan kendali yang sudah ada sebelumnya yaitu jaringan CAN (*Control Area Network*). Jaringan LIN ini secara implementasi hardware relatif lebih mahal.

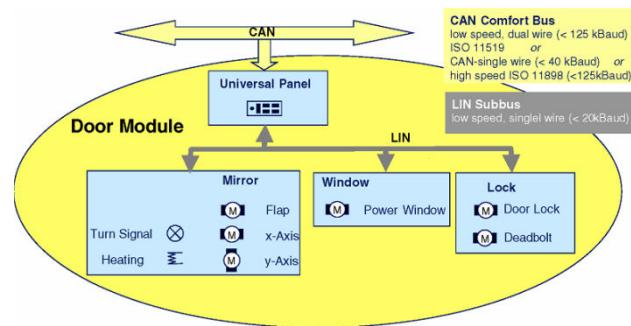
LIN pada dasarnya adalah jaringan serial master-slave, di mana 1 master bisa menangani sampai 15 slave. Dalam mode jaringan master-slave ini, hanya master yang bisa berinisiatif melakukan komunikasi dan hanya 1 slave yang bisa merespon sesuai identitas pesan/data yang dikirimkan. Kecepatan data di jaringan LIN bida sampai 20 kpbs dengan panjang kabel jaringan 40 meter (LIN versi 2.2).



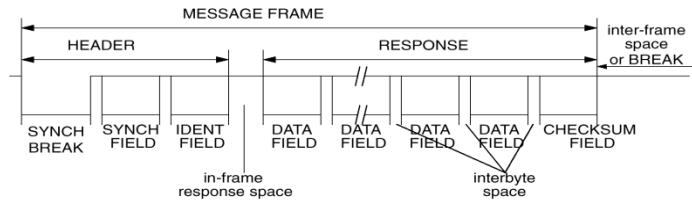
Gambar 3.79 Aplikasi LIN

Sebelum dikembangkan LIN, sistem jaringan kendali di otomotif menggunakan bus CAN. Semua sinyal kendali dikirimkan atau dibaca dari dan ke komputer mobil (ECU - *Engine Control Unit*) ke semua aktuator dan sensor-sensor, yang hampir semuanya menggunakan jaringan bus CAN. Kemudian dikembangkan sistem kendali yang tidak terpusat, dan sub-jaringannya dikembangkanlah LIN. Dan ini memberikan beberapa keuntungan, yang diantaranya menekan harga.

LIN merupakan jaringan kabel tunggal (*single wire*). Sehingga komunikasi akan terjadi secara bergantian, dan hanya master yang bisa berinisiatif untuk memulai komunikasi. Setiap frame data LIN terdiri atas *header* yang terdiri atas *synchbreak*, *synchfield* dan *identifier-field*. Kemudian diikuti dengan data (*data field*) yang bisa terdiri atas 2, 4, atau 8 byte dan diakhiri oleh *checksum*. Slave yang mempunyai *identifier-field* yang sama akan merespon pesan yang dikirimkan oleh master ini. Synchbreak pada dasarnya adalah bit 0 sebanyak 13 bit dan synchfield adalah 1 byte (0x55 atau b10101010).

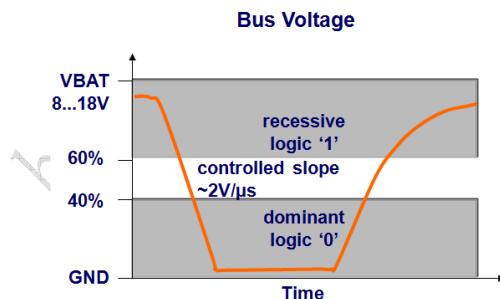


Gambar 3.80 Koneksi LIN dan CAN



Gambar 3.81 Frame Data LIN

UART/USART yang dimiliki oleh STM32F207 bisa diaktifkan pada mode LIN, walaupun untuk terhubung dengan jaringan LIN sesungguhnya masih perlu digunakan IC tambahan sebagai driver mengingat level logika jaringan LIN tidak kompatible dengan level logika GPIO STM32F207.

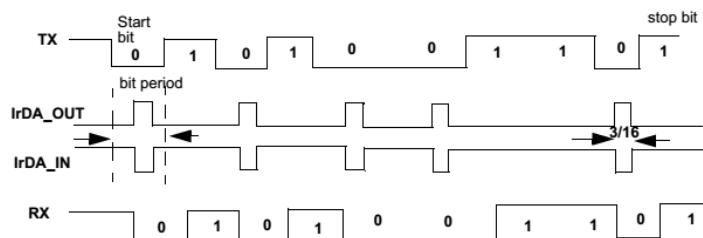


Gambar 3.82 Level Logika Jaringan LIN

3.6.2.4 Mode IrDA

Selain menggunakan kabel sebagai media transmisi, komunikasi serial bisa juga menggunakan media sinar infra merah (infra red). Salah satu contoh aplikasinya adalah fitur infra red yang ada di telepon genggam (handphone) yang bisa dipakai untuk mengirim dan menerima foto. Aplikasi infra red serial ini berbeda dengan aplikasi infra red yang dipakai oleh remote control televisi. Komunikasi serial dengan infra red merupakan komunikasi 2 arah bergantian (half duplex) dan dengan jarak yang dekat sekitar 1 meter. Berbeda dengan remote control TV yang merupakan komunikasi 1 arah dengan jarak yang cukup jauh, sehingga untuk menghindari gangguan dari sumber infra red yang lain, infra red dari remote dimodulasi di frekuensi 38 KHz atau 40 KHz. Kecepatan data bisa sampai 115.2 Kbps.

Walaupun tidak dimodulasi, bit-bit data sinyal IrDA tidak sama dengan keluaran USART yang menggunakan *non return to zero* (NRZ), di mana bit "1" adalah tegangan logika tinggi (5V misalnya), sedangkan bit "0" adalah tegangan logika rendah (0V). Sinyal IrDA telah distandardkan menggunakan sinyal *return to zero, inverted* (RZI) untuk bit-bit data yang akan dikirim ke pemancar infra merah (LED infra merah), sedangkan bit-bit data yang diterima dikodekan dengan sinyal *return to zero* (RZ). Sinyal RZI mengkodekan bit "0" sebagai pulsa positif dan bit "1" dikodekan dengan tanpa ada pulsa. Lebar pulsa diatur sekitar 3/16 perioda 1 bit.

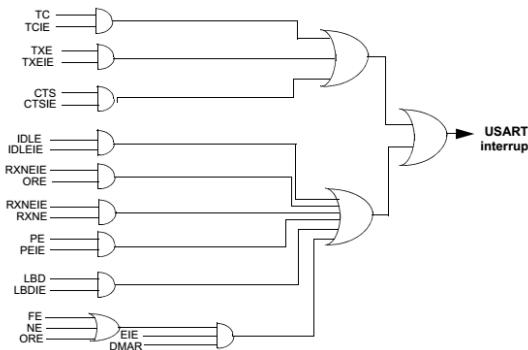


Gambar 3.83 Sinyal IrDA

3.6.2.5 Interupsi USART

USART bisa membangkitkan interupsi ketika:

1. Saat pengiriman: pengiriman selesai, interupsi CTS atau saat register data pengiriman kosong.
2. Saat penerimaan: deteksi jalur idle, error overrun, register data penerimaan kosong, error bit paritas, deteksi break LIN, flag noise, dan error framing.



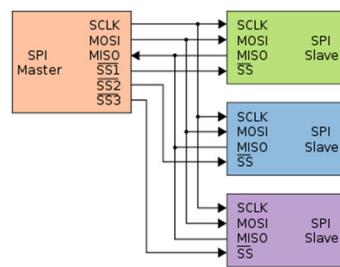
Gambar 3.84 Sumber-sumber Interupsi USART

3.7 SPI

SPI (*Serial Peripheral Interface*) merupakan sistem komunikasi serial sinkron yang pertama kali dikembangkan oleh Motorola, yang telah menjadi standar dan banyak digunakan untuk antarmuka ke memori (flash, RAM atau kartu SD dan MMC), sensor, LCD dan lain-lain. Oleh karena komunikasi sinkron, perlu dikirim sinyal clock untuk sinkronisasi. SPI sendiri merupakan komunikasi serial 4 kabel dan merupakan sistem komunikasi master dan slave. Adapun keempat kabel yang digunakan dalam komunikasi SPI adalah:

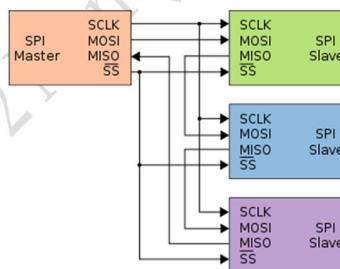
1. MOSI (*Master Out Slave In*), di sisi master merupakan output dan di sisi slave merupakan input.
2. MISO (*Master In Slave Out*), di sisi master merupakan input dan di sisi slave merupakan output.
3. SCK (*Clock*), merupakan clock sinkronisasi yang dikirimkan oleh master
4. SS (*Slave Select*), di sisi master merupakan output dan di sisi slave merupakan input, digunakan untuk mengaktifkan slave. Sinyal SS merupakan sinyal logika aktif rendah.

Dengan adanya pin SS, sebuah master bisa mengendalikan beberapa perangkat SPI dengan menggunakan pin SS yang berbeda.



Gambar 3.85 Sebuah Master Mengendalikan 3 buah Slave

Namun dengan koneksi khusus (*daisy chain*), sebuah master bisa mengendalikan beberapa slave hanya dengan 1 pin SS. Dalam koneksi khusus ini, pin MOSI master hanya terhubung dengan pin MOSI slave pertama, dan pin MISO master hanya terhubung ke pin MISO terakhir. Sedangkan pin MOSI slave kedua terhubung dengan pin MISO slave pertama dan seterusnya. Contoh aplikasi yang menggunakan model koneksi ini adalah SGPIO (Serial General Purpose Input/Output) dan JTAG.



Gambar 3.86 Koneksi Daisy Chain

I2S (Inter IC Sound) merupakan sistem antarmuka serial untuk peralatan audio yang dikembangkan pertama kali oleh Philip (sekarang NXP). I2S digunakan untuk mentransmisikan data audio digital dalam format PCM (Pulse Code Modulation) dari sebuah IC (misal dekoder MP3) ke IC lain (misal DAC). Dengan memisahkan jalur data dengan jalur clock, I2S akan memberikan jitter yang rendah

dibandingkan dengan sistem lain yang clock dan datanya disatukan. I2S sendiri merupakan komunikasi serial dengan 3 kabel:

1. Sinyal clock bit (CK)
2. Sinyal clock word, atau disebut juga word select (WS) atau left right clock (LRCLK), di mana 0 data merupakan sinyal audio untuk sinyal kiri dan saat 1 data merupakan sinyal audio kanan
3. Sinyal data (SD) itu sendiri, yang digunakan sebagai jalur data audio PCM yang mendukung frekuensi sampling sampai 192 kHz.

Selain itu I2S menyediakan pin master clock (MCK), yang walaupun bukan bagian dari sistem komunikasi I2S tetapi bisa digunakan untuk memberikan sinyal clock ke IC yang lain.

STM32F207 dilengkapi dengan 3 buah SPI (SPI1, SPI2 dan SPI3) dan 2 buah I2S (I2S2 dan I2S3). Fungsi dan pin yang digunakan untuk I2S hampir sama dengan SPI, oleh karena itu I2S dipetakan ke pin yang juga digunakan oleh SPI.

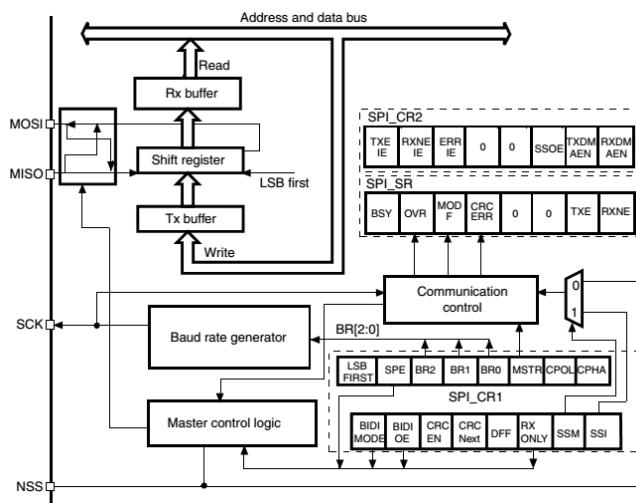
1. Pin SD dipetakan ke pin MOSI
2. Pin WS dipetakan ke pin SS
3. Pin CK dipetakan ke pin SCK

Oleh karena itu, SPI2 dan SPI3 tidak bisa berfungsi bersamaan dengan fungsi I2S2 dan I2S3. Ketika SPI aktif maka I2S tidak bisa digunakan, begitu juga sebaliknya. Selain itu, ketika I2S diprogram sebagai master, bisa mengeluarkan sinyal clock master (MCK) untuk perangkat (IC) audio eksternal, dengan frekuensi sampai $256 \times F_s$, di mana F_s adalah frekuensi sampling audio.

3.7.1 FUNGSI KERJA SPI

SPI mendukung komunikasi serial sinkron secara dupleks penuh maupun setengah dupleks. Panjang data bisa diprogram 8 bit atau 16 bit dengan kecepatan clock sampai $f_{PCLK}/2$. SPI bisa diprogram sebagai master atau pun sebagai slave. Pada saat jadi master, Pin NSS bisa diprogram untuk dikendalikan secara hardware (dengan menggunakan pin NSS yang telah dipetakan ke fungsi SPI) maupun secara software dengan menggunakan sembarang pin GPIO yang diprogram sebagai output. Dengan pengendalian secara software,

master bisa mengendalikan beberapa slave dengan menggunakan beberapa pin GPIO yang difungsikan sebagai pin NSS.



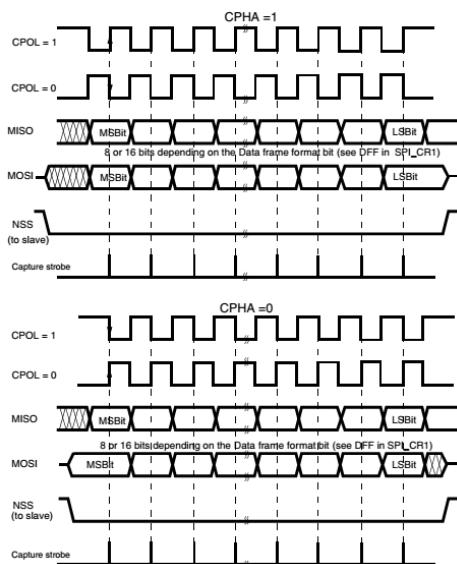
Gambar 3.87 Diagram Blok SPI

3.7.1.1 Register Kendali SPI

SPI dikendalikan oleh 2 register kendali (SPI_CR1 dan SPI_CR2), register data (SPI_DR) dan register status (SPI_SR). Data dikirim dan diterima melalui register data, yang terhubung ke pin MOSI dan MISO melalui sebuah register geser. Pengiriman dan penerimaan data bisa diprogram untuk mengirimkan bit MSB terlebih dahulu atau LSB dulu. SPI mempunyai 1 byte buffer pengiriman (*tx buffer*) dan 1 byte penerima data (*rx buffer*). Proses pengiriman data dilakukan dengan menuliskan data tersebut ke register SPI_DR, yang kemudian akan diteruskan ke tx buffer. Data yang diterima pertama akan disimpan di rx buffer, pembacaan SPI_DR akan membaca rx buffer dan menyimpannya di SPI_DR. Buffer ini dan juga proses pengiriman dan penerimaan data bisa dihubungkan dengan DMA. Pembangkit baud rate akan mengatur frekuensi clock, sementara sebuah master logic control akan mengendalikan pin NSS ketika pin ini dikendalikan secara hardware.

3.7.1.2 Polaritas dan Fase Clock

Selain kecepatan sinyal clock bisa diatur juga polaritas (CPOL) dan fase (CPHA). CPOL akan menentukan kondisi sinyal clock saat tidak ada aktifitas SPI (high atau low) sedangkan CPHA menentukan kapan data dilatch di pin MOSI atau dibaca di pin MISO berdasarkan transisi di sinyal clock. Saat CPHA=1, latch atau capture data akan terjadi di sisi kedua sinyal clock sedangkan saat CPHA=0 akan terjadi di sisi pertama sinyal clock. Sehingga latch dan capture data bisa terjadi saat transisi naik atau transisi turun sinyal clock (mengacu ke pengaturan CPOL). Pengaturan CPOL dan CPHA ini harus sama antara master dan slave.



Gambar 3.88 CPOL dan CPHA

3.7.1.3 Komunikasi Half-Duplex

SPI bisa juga diprogram untuk komunikasi half-duplex dengan 2 pin (clock dan data) dan ada 2 mode yang bisa dipilih:

1. 1 sinyal clock dan 1 sinyal data 2 arah
2. 1 sinyal clock dan 1 sinyal data 1 arah (hanya kirim atau hanya terima)

Pada mode pertama, data dikirim dan diterima melalui pin data secara bergantian. Master menggunakan pin MOSI dan slave menggunakan pin MISO. Saat proses pengiriman, pin data harus diatur sebagai output dan saat penerimaan diatur sebagai input. Pengaturan ini dilakukan melalui bit BIDIOE register SPI_CR1. Saat BIDIOE=1, pin data akan menjadi output dan saat BIDIOE=0 pin data menjadi input.

Pada mode kedua, SPI diprogram hanya di program untuk mengirim atau menerima. Ketika diprogram untuk mengirim saja, data dikirim lewat pin MOSI, ketika menjadi master, dan melalui pin MISO ketika menjadi slave. Sedangkan pin penerimaan, MISO saat jadi master dan MOSI saat jadi slave, bisa digunakan sebagai pin GPIO biasa. Begitu juga ketika diprogram hanya menjadi penerima, data diterima melalui pin MISO ketika menjadi master dan melalui pin MOSI ketika menjadi slave. Pin pengiriman bisa digunakan sebagai pin GPIO biasa.

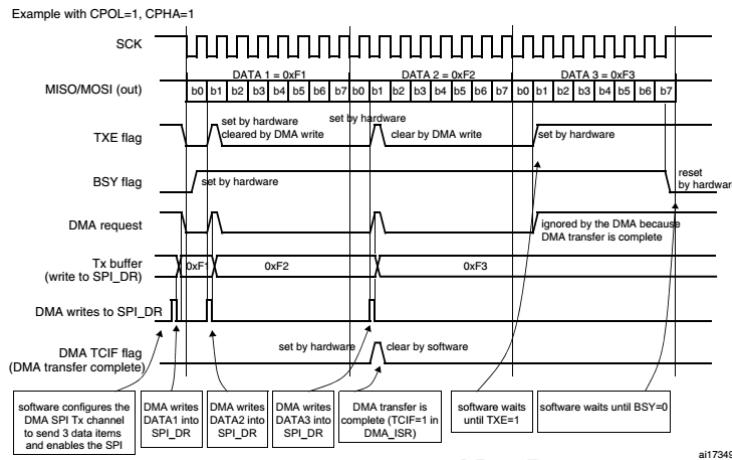
3.7.1.4CRC

SPI juga dilengkapi dengan perhitungan CRC (*Cyclic Redundancy Check*) yang bisa digunakan untuk meningkatkan kehandalan komunikasi melalui SPI. CRC ini dihitung menggunakan polinomial serial yang dapat diprogram dan mendukung 8 bit (CRC8) dan 16 bit (CRC16). CRC dipakai ketika mengirim maupun menerima data, byte-byte CRC ini disimpan di register CRC (SPI_RXCRCR dan SPI_TXCRCR).

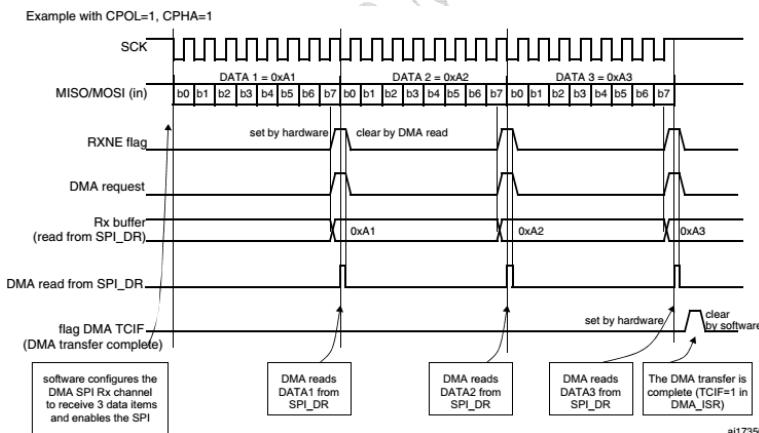
Pengiriman CRC dilakukan dengan men-set bit CRCNEXT di register SPI_CR1. Byte CRC dikirimkan setelah semua data dikirimkan. Begitu byte terakhir ditulikan ke register SPI_DR, bit CRCNEXT harus segera di-set. Setelah byte terakhir tersebut dikirimkan, maka byte CRC yang tersimpan di register SPI_TXCRCR langsung dikirimkan. Pada proses penerimaan, bit CRCNEXT harus di-set, setelah menerima data kedua terakhir, karena data terakhir yang diterima adalah CRC. Setelah CRC diterima, software bisa membandingkan antara CRC yang diterima dengan CRC yang dihasilkan oleh register SPI_RXCRCR.

3.7.1.5 Penggunaan DMA

SPI bisa digunakan bersama dengan DMA, sehingga SPI bisa mencapai kecepatan maksimumnya tanpa melibatkan CPU. DMA bisa diaktifkan baik untuk pengiriman maupun penerimaan. Pada pengiriman, permintaan DMA terjadi setiap kali bit TXE di-set dan saat penerimaan, permintaan DMA akan terjadi setiap kali bit RXNE di-set. Penggunaan DMA juga dimungkinkan hanya untuk pengiriman atau pun hanya penerimaan.



Gambar 3.89 Pengiriman Menggunakan DMA



Gambar 3.90 Penerimaan Menggunakan DMA

3.7.1.6 Status, Error dan Interupsi

Register status SPI_SR menyimpan bit-bit yang menunjukkan status dan error yang mungkin terjadi. Status dan error tersebut bisa juga menunjukkan interupsi yang dibangkitkan oleh SPI. Bit-bit status dan error ini perlu dibaca sebelum SPI digunakan untuk proses pengiriman atau juga penerimaan data.

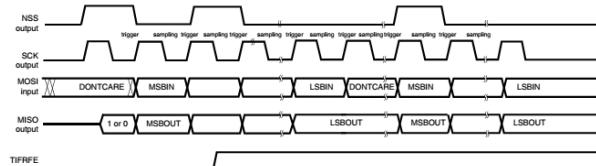
Bit status SPI terdiri atas:

1. BSY (*busy*), yang menunjukkan proses komunikasi (pengiriman atau penerimaan data) sedang berlangsung. Selama proses tersebut bit ini akan di-set (BSY=1), kecuali pada mode master atau mode penerimaan dua arah BSY tetap bernilai 0 saat penerimaan data. BSY bisa digunakan untuk apakah semua data telah dikirimkan atau belum dan juga berfungsi untuk mencegah tabrakan data pada mode multi master. BSY hanya bisa dimodifikasi oleh hardware, modifikasi secara software tidak akan berpengaruh.
2. TXE (*Tx buffer empty*), bila di-set TXE menunjukkan bahwa buffer TX telah kosong dan siap menerima data berikutnya. TXE akan dinolkan ketika ada penulisan ke registet data (SPI_DR).
3. RXNE (*Rx buffer not empty*), jika di-set menunjukkan bahwa ada data di buffer rx, artinya ada penerimaan data. Pembacaan register SPI_DR akan mereset bit RXNE.

Sedangkan error yang mungkin terjadi adalah

1. Kegagalan mode master (MODF, *master mode fault*), terjadi ketika pin NSS ter-pull down (ketika SPI menggunakan mode NSS secara hardware) atau bit SSI (ketika NSS dikendalikan secara software) bernilai nol. Error ini terjadi ketika SPI berfungsi sebagai master. Dalam mode multi master menunjukkan adanya konflik dengan master yang lain.
2. Kondisi overrun, terjadi ketika SPI menerima data sementara bit RXNE masih di-set, artinya data yang diterima sebelumnya belum dibaca.

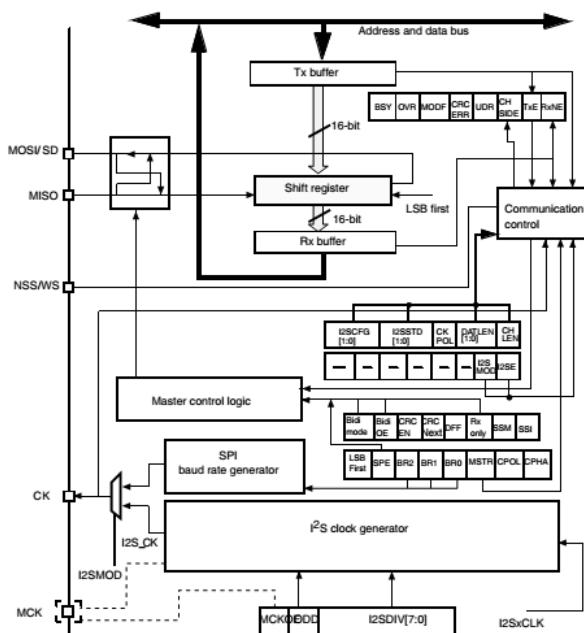
3. Error CRC, terjadi ketika fungsi CRC diaktifkan dan byte CRC yang diterima tidak sama dengan CRC yang dihasilkan oleh register CRC.
4. Error frame mode TI, terjadi ketika SPI menjadi slave dan ada pulsa di pin NSS saat komunikasi tengah berlangsung.



Gambar 3.91 Error Frame Mode TI

Semua bit status dan error tersebut bisa diprogram untuk membangkitkan interupsi SPI.

3.7.2 FUNGSI KERJA I2S



Gambar 3.92 Diagram Block I2S

Diagram blok I2S hampir sama dengan SPI, I2S bahkan dipetakan ke GPIO yang sama dengan SPI. Beberapa register kendali dan status digunakan bersama antara SPI dan I2S. Dan juga interupsi I2S juga 1 vektor dengan SPI. I2S bisa diprogram sebagai master atau slave.

3.7.2.1. Format Audio

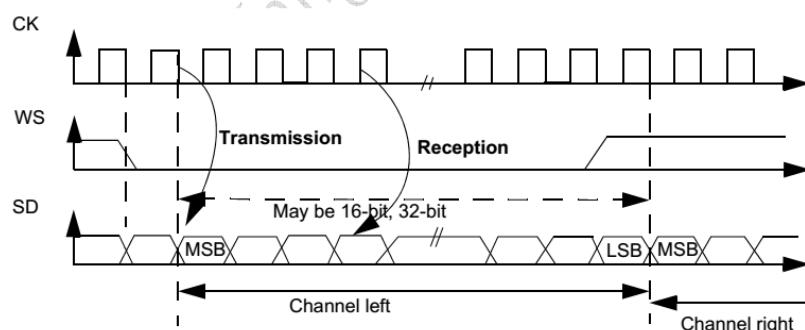
STM32F207 mendukung standar I2S:

1. Standar I2S Philips
2. Standar MSB
3. Standar LSB
4. Standar PCM

Dan juga mendukung hampir semua frekuensi yang digunakan di standar audio (8 kHz, 16 kHz, 22.05 kHz, 44.1 kHz, 48 kHz dan lain-lain). Format data bisa menggunakan 16, 24 atau 32 bit dengan ukuran 16 (hanya untuk format 16 bit) atau 32 bit.

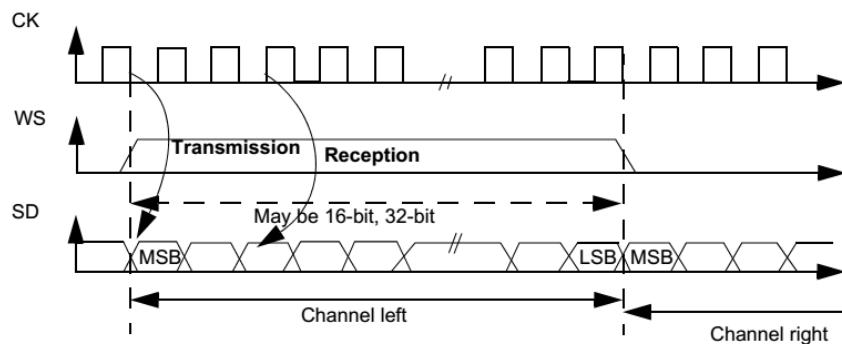
Standar-standar itu pada dasarnya adalah cara bagaimana data audio dikirimkan atau diterima. Selain standar PCM, sinyal WS digunakan untuk menunjukkan data dari saluran mana, kiri (*Left*) atau kanan (*Right*), yang sedang dikirimkan. Pada saat WS berlogika rendah, berarti data yang sedang dikirimkan berasal dari saluran L, sedangkan jika berlogika tinggi data audio berasal dari saluran R. Data dari saluran L selalu dikirimkan pertama kali. Oleh karena register data (SPI_DR, sama dengan SPI) berukuran 16 bit, saat format data 24 atau 32 bit, diperlukan 2 kali siklus baca atau tulis.

Pada standar I2S Philips sinyal WS ini diaktifkan 1 clock sebelum bit MSB dari data audio dikirimkan.



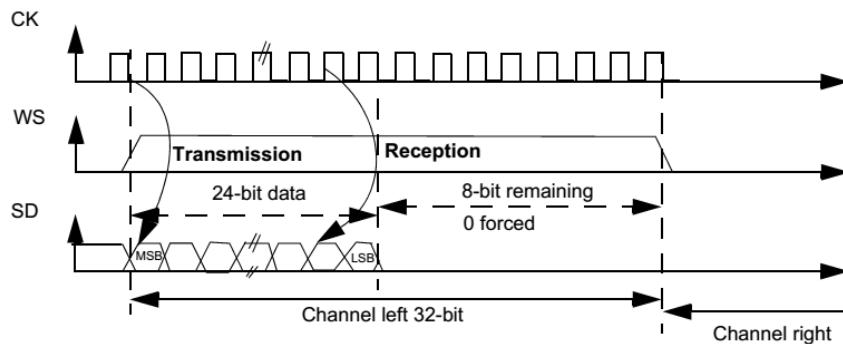
Gambar 3.93 Protokol I2S Philips 16/32 bit

Pada standar MSB dengan ukuran data sama dengan ukuran frame (misal data 16 bit dan frame 16 bit atau data 32 bit dan frame 32 bit) sinyal WS diaktifkan bersamaan dengan dikirimkannya bit MSB dari data audio.



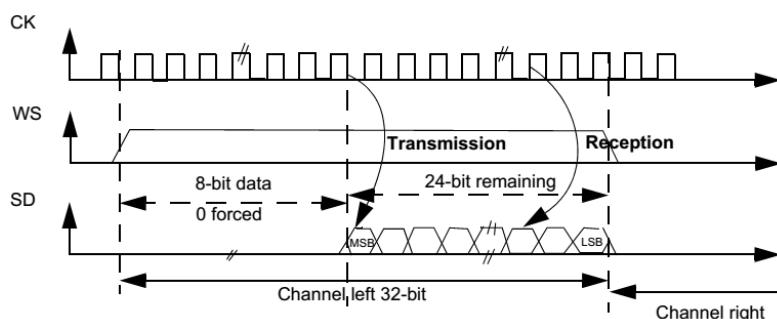
Gambar 3.94 Protokol MSB 16/32 bit dengan Frame 16 atau 32 bit

Pada saat transmisi data menggunakan ukuran frame yang lebih besar dari ukuran bit data, misal data 24 bit dan frame 32 bit, maka data akan diratakan (*justified*) ke MSB.



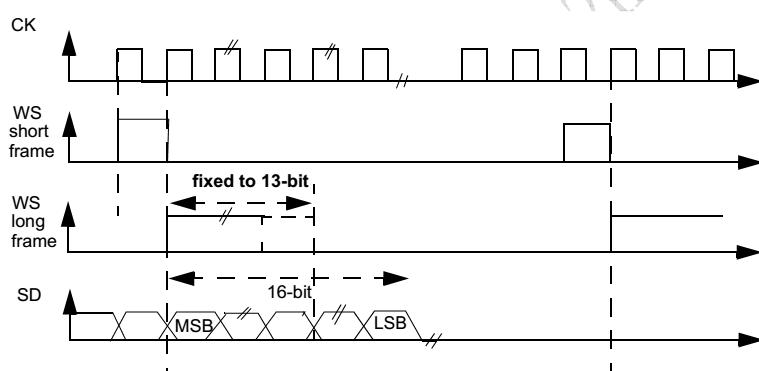
Gambar 3.95 Protokol MSB Format Data 24 bit dan Frame 32 bit

Sedangkan pada standar LSB, saat ukuran data sama dengan ukuran frame, diagram pewaktunya akan sama dengan standar MSB. Pada saat ukuran data berbeda dengan frame, misal data 24 bit dan frame 32 bit, maka data akan diratakan ke LSB.



Gambar 3.96 Protokol LSB Format Data 24 bit Frame 32 bit

Untuk standar PCM (*Pulse Code Modulation*), tidak diperlukan informasi saluran dari data yang dikirimkan. Sinyal WS digunakan untuk sinkronisasi data PCM, mode frame pendek dan frame panjang. Pada frame pendek, sinkronisasi hanya dilakukan satu kali setiap frame data audio (satu clock sebelum MSB), sedangkan pada sinkronisasi frame panjang, sinyal WS akan mempunyai lebar pulsa sebesar 13 bit dari frame audio.



Gambar 3.97 Standar PCM 16 bit, Frame 32 bit

3.7.2.2. Pembangkit Clock

Dalam I2S, kecepatan bit (*bit rate*) adalah aliran data di pin data dan frekuensi sinyal clock.

Bit rate I2S = panjang data (bit) x jumlah saluran x sampling frekuensi audio (F_s).

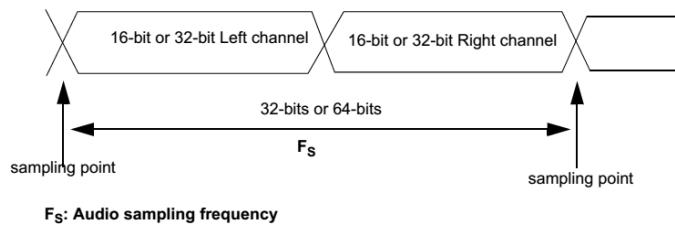
Misal, sinyal audio 16 bit stereo (2 saluran) dalam frame 16 bit dihitung dengan rumus:

$$\text{Bit rate I2S} = 16 \times 2 \times F_S$$

Jika frame yang digunakan 32 bit maka bit rate I2S akan menjadi:

$$\text{Bit rate I2S} = 32 \times 2 \times F_S$$

Di mana F_S bisa 96 kHz, 48 kHz, 44.1 kHz, 32 kHz, 22.05 kHz, 16 kHz, 11.025 kHz atau 8 kHz. Agar mencapai frekuensi sampling yang diinginkan, diperlukan pembagi clock yang dapat diprogram. Tabel 100 di *Reference Manual* menunjukkan nilai-nilai frekuensi sampling dan nilai-nilai pembagi beserta prosentasi errornya.



Gambar 3.98 Definisi Frekuensi Sampling Audio

3.7.2.3. Operasi Master dan Slave

I2S bisa beroperasi sebagai master atau slave. Pada saat menjadi master pin WS dan CK akan menjadi output, karena I2S akan mengendalikan pengiriman dan penerimaan data, yang berarti sinyal WS dan CK akan dikeluarkan oleh STM32F207. Sinyal MCK (clock master), bisa diakifkan atau tidak.

Pada saat menjadi slave, sinyal WS dan CK akan menjadi input, pengiriman dan penerimaan data akan dikendalikan oleh I2S master eksternal. I2S tidak akan mengeluarkan sinyal WS atau pun CK.

3.7.2.4. Status dan Interupsi

I2S mempunyai sistem status, error dan interupsi yang hampir sama dengan SPI. I2S mempunyai bit CHSIDE yang tidak dimiliki oleh SPI. Bit ini menunjukkan data dari saluran mana (kiri atau kanan) yang sedang dikirimkan atau diterima. Jika CHSIDE=0, data berasal dari saluran kiri, sedangkan jika CHSIDE=1, data berasal dari saluran kanan.

I2S hanya mempunyai kondisi error yaitu kondisi *underrun* (UDR) dan kondisi *overrun* (OVR). Kondisi UDR terjadi ketika terdeteksi clock pertama tetapi software belum melakukan penulisan ke register SPI_DR. Sedangkan kondisi OVR terjadi ketika penerimaan data tetapi data sebelumnya belum dibaca.

3.8 I2C

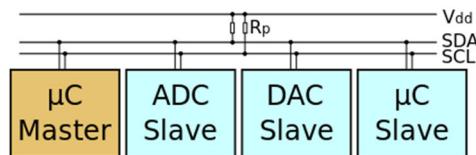
I2C atau *Inter Integrated Circui* merupakan standar antarmuka komunikasi serial sinkron 2 kabel (data dan clock) antar IC (misal antara mikrokontroler dengan EEPROM) dan biasanya hanya untuk komunikasi jarak dekat, biasanya dalam PCB yang sama. I2C pertama kali dikembangkan oleh Philips Semiconductor (atau sekarang NXP Semiconductor) pada tahun 1982. Pada awalnya Philips selaku penemu, meminta royalti kepada setiap pabrikan semikonduktor yang menerapkan standar I2C. Oleh karena itu, ATMEL standar yang bekerja secara kompatible dengan I2C dan dinamakan TWI (*Two Wire Interface*). Namun sejak Oktober 2006, Philip/NXP membebaskan royalti ini kecuali bagi yang ingin mendapatkan alamat slave unik yang dikeluarkan oleh NXP.

Pada tahun 1994, Intel dan Duracell mengembangkan *System Management Bus* (SMBus) sistem komunikasi yang berasal dari standar I2C dengan beberapa penambahan fitur. SMBus dikembangkan untuk sistem kendali bus dan manajemen daya (misal pengendalian sistem pengisian batere atau pengukuran suhu di PC). Kemudian dikembangkan lagi PMBus (*Power Management Bus*) yang dikhkususkan untuk manajemen daya. STM32F207 mendukung SMBus versi 2.0 dan juga PMBus.

Sama seperti SPI, I2C juga merupakan komunikasi master dan slave, bedanya I2C hanya menggunakan 2 kabel, yaitu data (SDA) yang bersifat 2 arah (*bidirectional*) dan clock sinkronisasi (SCL). Sinyal clock dikirimkan oleh master dan frekuensi clock menentukan kecepatan data. Sejak pertama diperkenalkan, frekuensi clock I2C sudah mencapai 5 MHz di versi 4 yang dikeluarkan tahun 2012. Frekuensi standar I2C adalah 100 kHz dan mode cepatnya (*fast mode*) adalah 400 kHz. Frekuensi 100 kHz dan 400 kHz-lah yang didukung oleh I2C STM32F207. Pin SDA dan SCL bersifat *open-drain*, oleh karena itu diperlukan resistor pull-up eksternal.

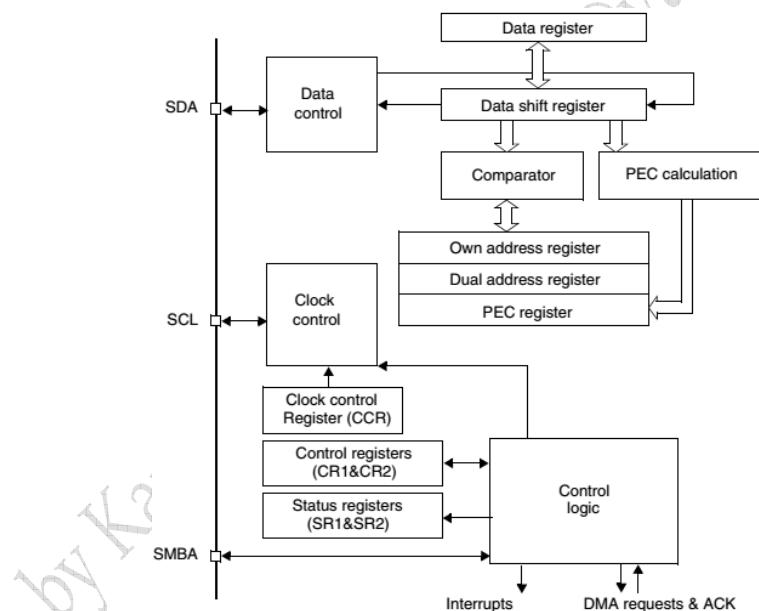
Perangkat I2C yang bertindak sebagai master bisa mengendalikan beberapa perangkat yang bertindak sebagai slave dengan menggunakan

alamat yang berbeda untuk setiap slave. I2C mendukung pengalaman 7 bit dan 10 bit, walaupun bisa juga menggunakan alamat 16 bit. Jumlah perangkat I2C yang bisa terkoneksi ke jaringan (*node*) dibatasi oleh kapasitansi total bus 400 pF, dengan jarak hanya beberapa meter.



Gambar 3.99 Contoh Koneksi Master-Slave I2C

3.8.1 FUNGSI KERJA I2C



Gambar 3.100 Diagram Blok I2C

Seperti telah dijelaskan, I2C adalah mode komunikasi 2 kabel yang terdiri atas pin SDA, yang bersifat 2 arah di mana data dikirim dan diterima, dan pin SCL untuk mengirimkan clock sinkrinisasi dari perangkat yang bertindak sebagai master. Kedua pin ini bersifat drain terbuka dan memerlukan resistor pull up eksternal. Kecuali untuk SMBus ada

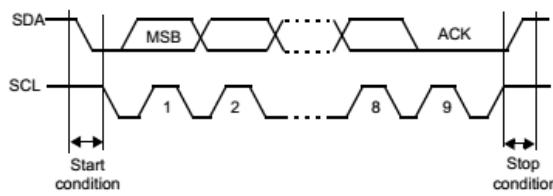
penambahan pin yaitu SMBA (*SMBus Alert*) seperti akan dijelaskan kemudian.

I2C dikendalikan oleh 2 buah register kendali (I2C_CR1 dan I2C_CR2), 2 buah register status (I2C_SR1 dan I2C_SR2), register clock (I2C_CCR), dan juga register data (I2C_DR). selain itu pada saat menjadi slave, I2C mempunyai 2 register alamat (*own address register*) yaitu register um I2C_OAR1 yang mendukung pengalamatan 10 bit dan register I2C_OAR2 (pengalamatan 7 bit).

I2C juga mendukung PEC (*Packet Error Checking*) yang berfungsi hampir sama dengan CRC yang dapat dikirimkan setelah byte terakhir data dikirimkan atau untuk pengecekan data saat menerima. PEC juga dipakai pada mode SMBus.

3.8.1.1 Mode Master dan Slave

Dalam I2C, komunikasi diawali oleh kondisi start yang dikirimkan oleh master dan diakhiri oleh kondisi stop yang juga berasal dari master. Data dan alamat dikirimkan per byte (8 bit) dengan bit MSB dikirimkan terlebih dahulu. Byte pertama setelah kondisi start biasanya adalah byte alamat (1 byte jika alamat 7 bit dan 2 byte jika menggunakan alamat 10 bit). Byte alamat selalu dikirimkan oleh master. Pada clock ke-9, slave akan mengirimkan sinyal ACK (*acknowledge*) sebagai tanda bahwa transmisi data telah diterima. Walaupun sinyal ACK bisa diaktifkan atau tidak.



Gambar 3.101 Protokol I2C

I2C bisa dioperasikan sebagai master maupun slave. Setelah reset, I2C berada di mode slave. Sebagai master, I2C akan membangkitkan sinyal clock, memulai (kondisi start) dan mengakhiri (kondisi stop) komunikasi. Sedangkan sebagai ketika berfungsi sebagai slave, akan mendeteksi kondisi start, dan membandingkan alamat yang diterima dengan alamat yang tersimpan di register I2C_OAR1 atau I2C_OAR2. Jika alamat yang

diterima sama, I2C akan mengirimkan sinyal ACK dan siap menerima data, jika tidak I2C akan mengabaikan kondisi start dan menunggu kondisi start berikutnya.

3.8.1.2 Penggunaan DMA

I2C bisa diprogram untuk mengirim dan menerima melalui DMA. DMA harus diinisialisasi dan diaktifkan sebelum transfer data. DMA diaktifkan dengan menge-set bit DMAEN di register I2C_CR2. Ketika pengiriman menggunakan DMA, data yang akan dikirimkan disimpan terlebih dahulu ke memori yang telah ditentukan melalui kendali DMA, kemudian akan dikirimkan ke register I2C_DR ketika bit TxE di-set. Sedangkan saat penerimaan menggunakan DMA, data yang diterima oleh register I2C_DR akan disimpan ke memori DMA yang telah ditentukan. Setiap akhir pengiriman atau penerimaan bisa membangkitkan interupsi.

3.8.1.3 Pengecekan Error Paket

Pengecekan error paket (PEC) merupakan fitur I2C untuk meningkatkan kehandalan komunikasi. PEC dihitung dengan menggunakan rumus:

$$C(x) = x^8 + x^2 + x + 1$$

yang merupakan polinomial CRC-8. PEC diaktifkan dengan menge-set bit ENPEC register I2C_CR1.

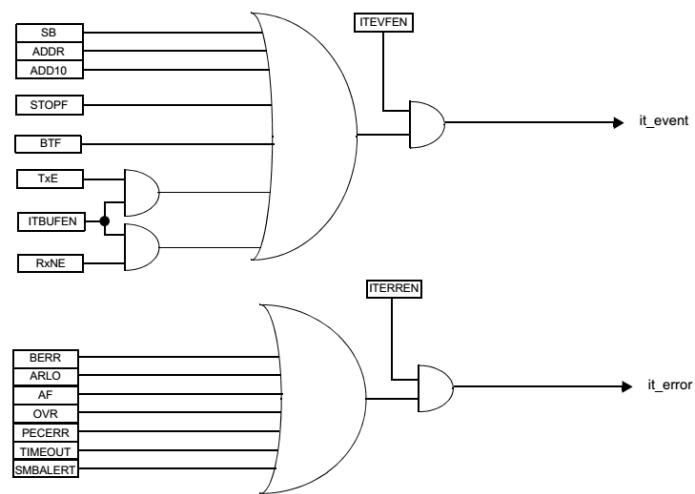
Pada proses pengiriman data I2C, PEC akan dikirimkan setelah byte data terakhir selesai dikirimkan. Dan pada proses penerimaan I2C, PEC akan diterima sebagai byte terakhir. Byte PEC ini disimpan di register I2C_SR2. Ketika menerima byte PEC yang tidak sesuai, I2C akan mengirimkan byte NACK.

3.8.1.4 Kondisi Error dan Interupsi

Ada beberapa kondisi error yang mungkin terjadi saat komunikasi I2C terjadi: error bus (BERR), error ACK (AF), error arbitrasi (ARLO) dan kondisi overrun/underrun (OVR). Kondisi BERR terjadi kalau terdeteksi kondisi start atau kondisi stop selama pengiriman byte alamat atau data. Ketika menjad slave, jika kondisi start yang diterima akan dianggap sebagai restart, sehingga I2C akan menunggu byte alamat atau kondisi stop. Jika kondisi stop yang diterima, I2C akan stop.

Error ACK terjadi ketika I2C mendeteksi bit *nonacknowledge* (NACK). Ketika ini terjadi slave harus membebaskan saluran I2C sedangkan master harus menghentikan atau membangkitkan kembali kondisi start secara software. Error ARLO terjadi ketika I2C mendeteksi kondisi kehilangan arbitrasi (*arbitration lost*). Hal ini bisa mengakibatkan I2C secara otomatis kembali ke mode slave. Sedangkan error OVR terjadi ketika data yang diterima tidak segera dibaca dari register DR sehingga akan tertimpas oleh byte data berikutnya.

Semua kondisi error bisa membangkitkan interupsi, selain itu juga ada beberapa kondisi yang bisa membangkitkan interupsi seperti ditunjukkan oleh gambar.



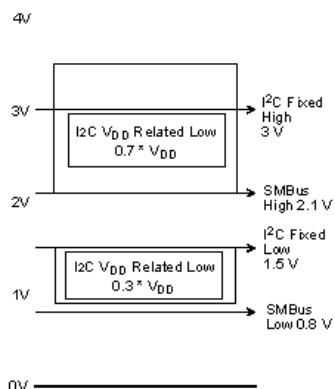
Gambar 3.102 Sumber Interupsi I2C

3.8.2 FUNGSI SMBUS

SMBus atau System Management Bus, seperti telah dijelaskan, merupakan salah satu pengembangan I2C oleh perusahaan Intel. SMBus ini digunakan untuk kendali bus sistem atau manajemen daya, misal mengendalikan sistem pengisian batere di laptop. SMBus digunakan untuk memberikan atau membaca perintah ke atau dari sebuah perangkat. Dalam sistem SMBus, perangkat bisa berfungsi sebagai slave yang akan menerima dan merespon sebuah perintah, master yang mengirim perintah, membangkitkan clock dan mengakhiri proses pengiriman, dan host yang pada dasarnya master dengan fungsi khusus

yang akan menjadi penghubung utama dengan CPU pengendali sistem. Host harus bisa menjadi master atau slave dan harus mendukung protokol notifikasi yang biasa digunakan oleh SMBus.

Ada beberapa perbedaan antara I²C standar dengan SMBus. Kecepatan SMBus dibatasi dengan frekuensi clock 10 kHz - 100 kHz (SMBus versi 2.0), sedangkan I²C standar frekuensi clock maksimum di 400 kHz dengan tidak ada batasan frekuensi clock minimum. Oleh karena frekuensi clock minimal 10 kHz, SMBus mengenal adanya timeout sebesar 35 milidetik. Artinya sinyal clock tidak boleh berada di kondisi statik lebih dari 35 milidetik. Perbedaan selanjutnya adalah level logika untuk SMBus telah ditentukan di 0.8V untuk kondisi logika rendah dan di 2.1V untuk logika tinggi, sedangkan I²C level logika ditentukan oleh tegangan yang digunakan (VDD).



Gambar 3.103 Perbedaan Level Logika I²C dengan SMBus

Jika alamat perangkat I²C adalah tetap (statis), 7 atau 10 bit, SMBus mendukung alamat dinamis melalui fungsi/protokol ARP (*Address Resolution Protocol*). Untuk bisa menerapkan fungsi ARP ini, perangkat SMBus harus mempunya identitas yang unik atau UID (*Unique Device Identifier*). Selain itu SMBus, seperti disebutkan di spesifikasi versi 2.0, mendukung 9 protokol, yang pada dasarnya protokol untuk pengiriman perintah, pengiriman dan penerimaan data.

SMBus juga mempunyai sinyal tambahan selain data dan clock, yaitu SMBus Alert (SMBA). Sinyal ini bisa digunakan untuk menginterupsi master sehingga master bisa segera merespon, dengan membuat sinyal SMBA berlogika rendah. Master akan merespon dengan mengirimkan

alamat *Alert Response Address* (ARA). Perangkat yang tadi mengirimkan sinyal SMBA kemudian balik merespon ke master. Jika lebih dari satu perangkat mengirimkan sinyal ARA secara bersamaan, maka perangkat yang mempunyai prioritas lebih tinggi (yaitu perangkat dengan alamat SMBus terendah) yang akan merespon terlebih dahulu. Perangkat yang sudah merespon ARA, harus mematikan kembali sinyal SMBA (logika tinggi). Jika master masih melihat sinyal SMBA, maka master akan kembali mengirimkan sinyal ARA.

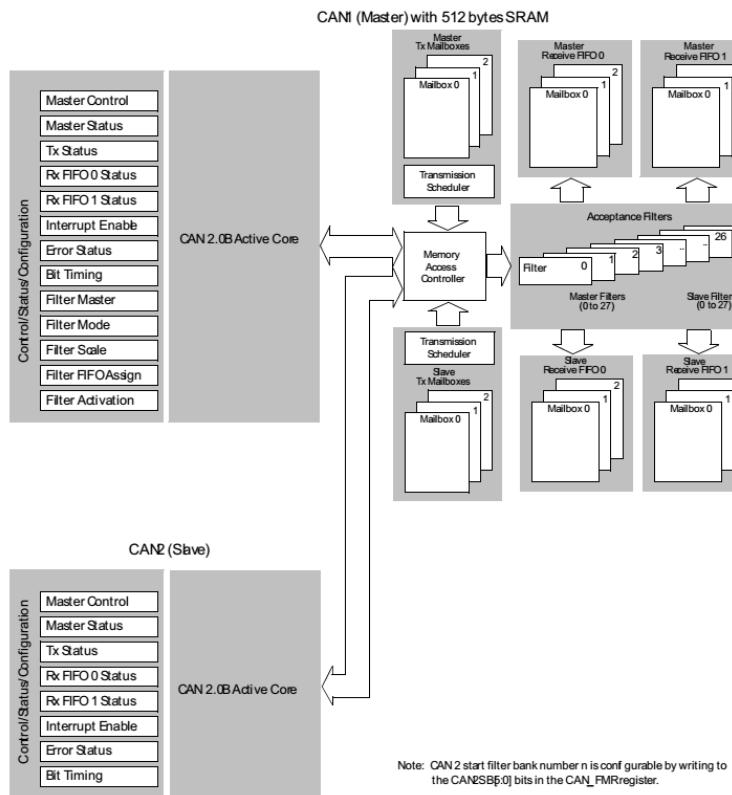
Register-register kendali, data dan status SMBus dan juga pemetaan pin sama dengan I2C, termasuk juga error dan interupsi. Fungsi SMBus diaktifkan dengan men-set bit SMBUS di register I2C_CR1.

3.9 CAN

CAN atau *Control Area Network* merupakan protokol komunikasi yang dikembangkan terutama untuk sistem komunikasi di kendaraan. CAN pertama kali dikembangkan pada tahun 1983 oleh Robert Bosch GmbH sebuah perusahaan Jerman dibidang komponen otomotif, terutama untuk sistem kelistrikan dan sistem elektronik kendaraan. Protokol CAN pertama kali dirilis saat konferensi *Society of Automation Engineers* (SAE) yang diselenggarakan di Detroit, Michigan pada tahun 1986. Setahun kemudian Philips (sekarang NXP) membuat chip CAN pertama, dan BMW seri 8 merupakan mobil pertama yang menerapkan protokol CAN.

Bosch merilis CAN versi terakhir (versi CAN 2.0) pada tahun 1991, yang mempunyai 2 bagian, yaitu bagian A untuk perangkat CAN dengan pengenal 11 bit (*CAN standar*) dan bagian B untuk perangkat CAN dengan pengenal 19 bit (*extended*). Perangkat CAN dengan pengenal 11 bit biasa disebut dengan CAN 2.0A dan perangkat dengan pengenal 29 bit disebut CAN 2.0B. Pada tahun 1993, organisasi standar internasional (ISO) merilis CAN ISO 11898 dalam 2 bagian: ISO 11898-1 yang mendefinisikan link data (*data link layer*) dan ISO 11898-2 yang mendefinisikan *physical layer* untuk kecepatan tinggi. Pada tahun 1993, ISO merilis ISO 11898-3 untuk CAN kecepatan rendah. Dua standar ISO terakhir bukan bagian dari standar CAN yang dikeluarkan oleh Bosch. Selain digunakan untuk sistem komunikasi internal di sistem kendaraan, CAN juga digunakan sebagai salah satu dari 5 protokol yang digunakan dalam sistem pengecekan kendaraan atau OBD (*On Board Diagnostic*).

3.9.1 FUNGSI KERJA CAN

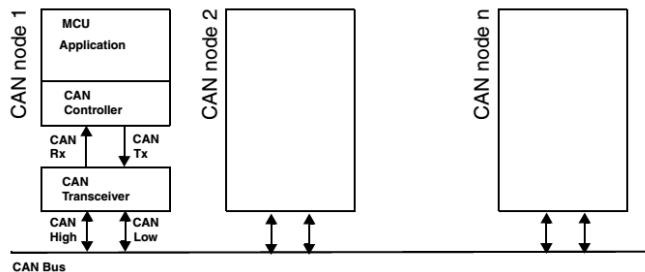


Gambar 3.104 Diagram Blok CAN

STM32F207 dilengkapi dengan 2 buah periperal CAN (CAN1 dan CAN2) yang mendukung CAN versi 2.0 A dan B, ST Micro menamakan perangkat CANnya sebagai bxCAN (*basic extended CAN*). CAN1 bertindak sebagai master yang bisa mengatur komunikasi antara CAN slave (CAN2) dengan SRAM 512 byte. Kedua CAN akan memakai secara bersama-sama memori tersebut, walau pun CAN2 tidak bisa mengakses secara langsung.

Sesuai dengan namanya, *basic extended*, periperal CAN mendukung protokol CAN dasar (*basic*) atau CAN 2.0A dan CAN pengembangan (*extended*) atau CAN 2.0B aktif maupun pasif dengan kecepatan data sampai 1 Mbps. Perangkat CAN juga mendukung protokol *time-triggered CAN* (TTCAN). Pada saat TTCAN diaktifkan, TTCAN mendukung

pengiriman ulang pesan secara otomatis dan akan mengganti penanda waktu (*time stamp*) di 2 byte data terakhir dari paket pesan CAN. TTCAN bisa digunakan agar aplikasi software bisa menggunakan perangkat CAN untuk aplikasi *hard real time*.



Gambar 3.105 Topologi Jaringan CAN

CAN dilengkapi dengan.

Bab 4

SISTEM MINIMUM STM32F207

Sebuah mikrokontroler harus didukung oleh perangkat atau komponen eksternal agar bisa bekerja dengan baik. Komponen eksternal minimum yang diperlukan agar mikrokontroler bisa bisa bekerja dengan baik dinamakan dengan sistem minimum. Setiap mikrokontroler mempunyai sistem minimum yang berbeda-beda, walaupun sebenarnya diberi catu daya (tegangan) saja mikrokontroler sudah bisa disebut dengan sistem minimum karena komponen-komponen untuk membentuk sebuah sistem minimum sudah tersedia di dalam mikrokontroler itu sendiri: CPU, memori program (ROM) dan memori data (RAM). Mikrokontroler yang memerlukan sistem minimum yang lebih kompleks juga.

ST Microelectronics, selaku pembuat mikrokontroler STM32F207, telah mendokumentasikan apa saja yang harus diperhatikan agar STM32F207 bisa bekerja dengan optimum dalam sebuah catatan aplikasi yang berjudul: *AN3320 Application Note: Getting started with STM32F2xxx/21xxx MCU hardware development*. Dalam catatan aplikasi tersebut ada beberapa hal yang harus diperhatikan ketika

4.1 CATU DAYA

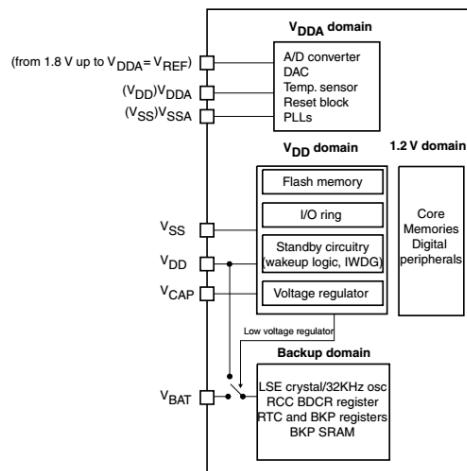
Sebuah mikrokontroler tentunya memerlukan catu daya (*power supply*) untuk bekerja sesuai dengan rancangannya. STM32F207 sendiri dirancang untuk berkerja di tegangan 1.8V sampai 3.6V DC, kecuali untuk kemasan WLCSP yang bisa bekerja di tegangan 1.65V sampai 3.6V. Di kebanyakan aplikasi, STM32F2 diberi catu daya di tegangan 3.3V DC. Tegangan di bawah tegangan minimum tidak akan membuat

mikrokontroler bekerja dengan baik sedangkan tegangan di atas tegangan maksimum akan merusak mikrokontroler. Selain itu, sumber tegangan juga harus menyediakan arus yang dibutuhkan oleh mikrokontroler yang besarnya tergantung mode kerja dan periperal yang diaktifkan.

4.1.1 SKEMA CATU DAYA STM32F207

Untuk fitur-fitur yang dimilikinya, STM32F207 dirancang dengan skema atau pembagian sistem tegangan sebagai berikut:

1. Tegangan 1.2V (internal)
2. Tegangan VDD (digital)
3. Tegangan VDDA (analog)
4. Tegangan cadangan VBAT (*backup*)

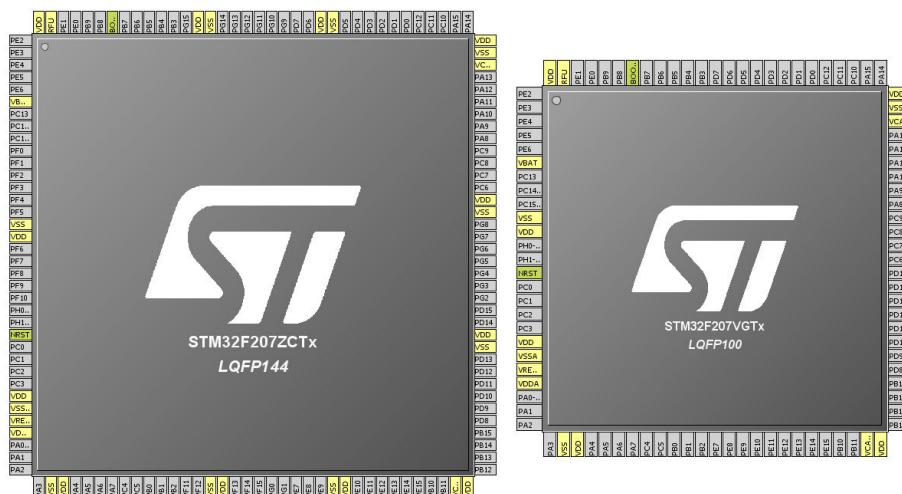


Gambar 4.1 Skema Catu Daya STM32F207

Tegangan 1.2V merupakan tegangan yang berasal dari regulator tegangan internal. Tegangan ini akan mentenagai *core*, memori (RAM) dan periperal digital. Tegangan VDD merupakan tegangan masukan yang berasal dari luar yang akan mentenagai pin I/O, memori flash, watchdog, sistem *wake-up* dan sumber untuk regulator internal 1.2V. Tegangan VDDA digunakan untuk mentenagai bagian analog (ADC dan DAC), sistem reset, PLL dan sensor suhu internal. Sedangkan tegangan cadangan (VBAT) digunakan untuk mentenagai sistem clock frekuensi rendah, RTC dan SRAM BKP (*back up*). VBAT mempunyai saklar pemilih

internal, jika tegangan utama (VDD) masih aktif, maka tegangan cadangan diambil dari VDD, tetapi jika tegangan utama mati, maka tegangan cadangan diambil dari pin VBAT. Tegangan ini biasanya menggunakan sumber batere (batere koin) atau kapasitor super, terutama ketika fitur RTC diaktifkan.

Konfigurasi pin catu daya STM32F207 ditentukan oleh kemasannya. Biasanya setiap kemasan memiliki pin catu daya digital lebih dari satu, sedangkan pin catu daya analog hanya ada satu. Semua pin catu daya tersebut harus dihubungkan ke sumber tegangan sesuai dengan spesifikasi yang telah ditentukan.



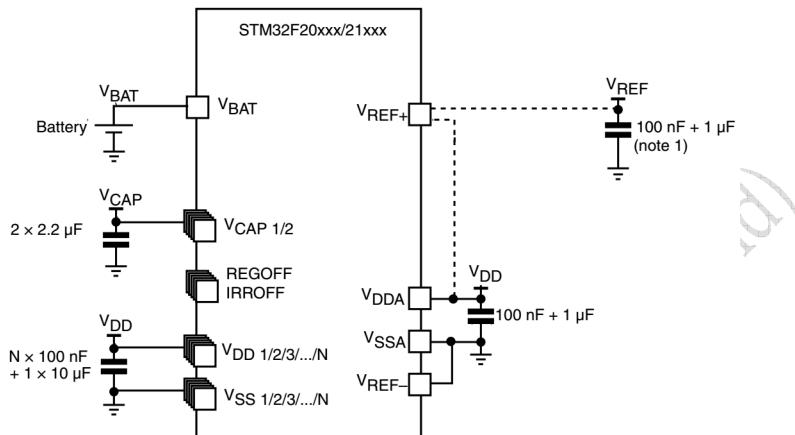
Gambar 4.2 Konfigurasi Pin Power di Kemasan LQFP100 dan LQFP144

4.1.2 REKOMENDASI RANCANGAN

Setiap pin catu daya harus dihubungkan dengan sumber tegangan sesuai dengan spesifikasi yang telah ditentukan. Untuk masing-masing tegangan (digital dan analog) harus dihubungkan dengan satu kapasitor *decoupling* (minimal 4,7 μ F) dan semua pin VDD dan VDDA harus diberi kapasitor decoupling kapasitor keramik dengan nilai 100 nF. Di dalam rancangan PCB, semua kapasitor tersebut harus diletakan sedekat mungkin dengan pin catu daya. Sedangkan VBAT bisa dihubungkan dengan sebuah batere, kapasitor super.

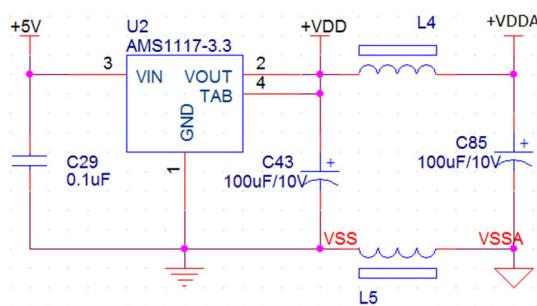
STM32F2 juga mempunyai pin VCAP1 dan VCAP2 yang terhubung dengan regulator internal. Pin VCAP ini disarankan untuk dihubungkan

dengan kapasitor decoupling dengan nilai 2.2 μF , untuk menstabilkan dan menghilangkan noise dikeluaran regulator internal.



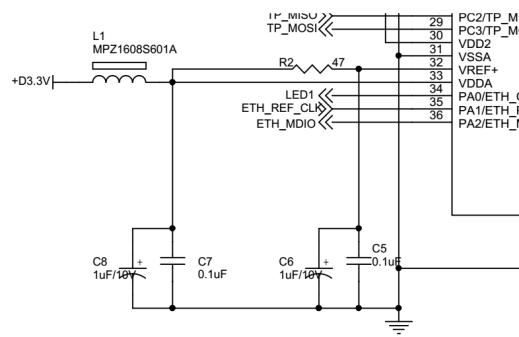
Gambar 6.1 Rekomendasi Catu Daya

Ketika fungsi analog diaktifkan (ADC atau DAC), sumber tegangan untuk analog bisa dipisahkan melalui sebuah ferit untuk mengurangi noise yang ditimbulkan oleh bagian digital agar tidak mempengaruhi bagian analog. Biasanya hanya VDDA yang dipisahkan dengan VDD dengan sebuah ferit, sedangkan VSSA langsung dihubungkan dengan VSS.



Gambar 6.2 Rekomendasi Tegangan Analog

Tegangan acuan untuk analog (VREF) dihubungkan dengan VDDA melalui sebuah resistor 47 Ohm. Tujuannya juga sama untuk mengurangi noise agar didapat akurasi ADC dan DAC yang lebih baik.



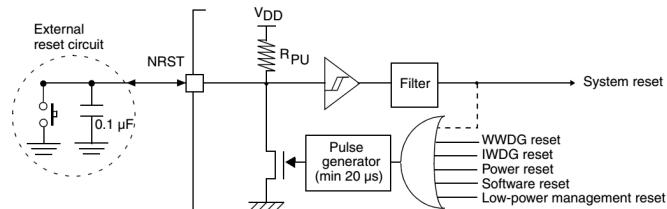
Gambar 6.3 Rekomendasi Untuk VREF

4.2 SISTEM RESET

Sistem reset diperlukan mikrokontroler agar saat pertama kali dinyalakan mikrokontroler berada dikondisi yang diketahui, mode kerja dan semua periperal berada di kondisi default (*power on reset*). Selain itu reset juga diperlukan saat program yang dijalankan mengalami *hang*. Reset akan mengembalikan mikrokontroler ke keadaan semula dan memulai program dari awal.

Sumber reset STM32 berasal dari:

1. Eksternal melalui pin NRST, aktif rendah.
2. Reset dari detektor tegangan
3. Reset dari timer watchdog ketika overflow (WWDG dan IWDG)
4. Reset secara software
5. Reset dari manajemen catu daya

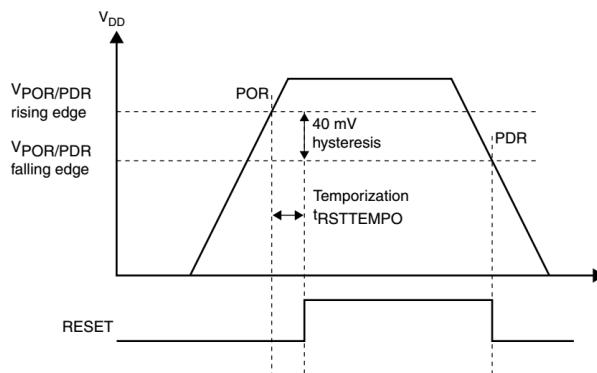


Gambar 6.4 Sumber Reset STM32F207

STM32F207 tidak membutuhkan rangkaian reset eksternal khusus, hanya sebuah kapasitor pull down di pin NRST dengan nilai 100 nF untuk melindungi STM32 agar tidak tereset oleh karena tegangan parasitik

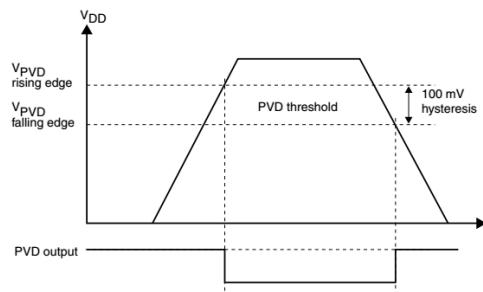
akibat noise. Sebuah saklar bisa ditambahkan agar STM32F207 bisa direset secara manual.

STM32F207 dilengkapi dengan detektor tegangan internal. Detektor ini akan membaca tegangan catu daya pada saat pertama kali dinyalakan. Detektor ini dinamakan dengan POR/PDR (*Power On Reset/Power Down Reset*). POR/PDR akan memastikan pada saat catu daya pertama kali dinyalakan, CPU tetap berada di kondisi reset sampai tegangan catu daya stabil. Karena apabila tegangan belum stabil (berada di bawah tegangan minimal yang diperlukan), CPU bisa berada di kondisi yang tidak bisa diprediksi, karena salah membaca data dari memori flash. Kondisi ini dikenal dengan kondisi *brown-out*.



Gambar 6.5 Detektor Tegangan POR/PDR

Tegangan ambang detektor tegangan bisa diprogram melalui sebuah *Programmable Voltage Detector* (PVD) melalui register PWR_CR (*Power Control Register*). Pada saat PVD digunakan, detektor tegangan akan membaca tegangan di pin catu daya dan membandingkannya dengan nilai ambang yang disimpan di bit PLS[2:0] register PWR_CR. PVD juga bisa diprogram membangkitkan interupsi kalau terdeteksi tegangan di bawah nilai ambang. Sehingga program bisa melakukan tindakan untuk menangani catu daya yang tiba-tiba mati, misal menyimpan data ke memori (EEPROM).



Gambar 6.6 Nilai Ambang PVD

Sumber reset selanjutnya adalah timer watchdog (WWDG dan IWDG). Timer watchdog pada dasarnya adalah timer yang setelah diaktifkan akan melakukan cacahan naik. Ketika terjadi overflow timer watchdog akan memberikan sinyal reset kepada CPU. Watchdog digunakan agar pada saat program menjadi hang akan mereset CPU dan program akan kembali mulai dari awal. Pada saat berjalan normal, program akan secara berkala mereset watchdog agar tidak overflow. Ketika program hang, fungsi untuk mereset watchdog juga ikut hang, sehingga watchdog akan mengalami overflow dan mereset CPU.

Reset juga bisa dibangkitkan pada saat STM32F207 memasuki mode stand by atau mode stop. Mode ini diatur melalui byte pilihan (*option byte*). Pada saat diaktifkan, ketika mode stand by atau mode stop aktif STM32F207 akan berada di kondisi reset, bukan di kondisi stand by atau stop.

Sumber reset yang terakhir adalah reset yang dibangkitkan oleh software. Reset dilakukan dengan dilakukan dengan memprogram register AIRCR, seperti dijelaskan di sub bab 2.8.4.

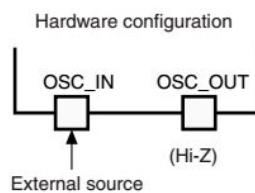
4.3 CLOCK

STM32F207 memerlukan 2 buah jenis clock, clock dengan frekuensi tinggi untuk core CPU (SYSCLK), sistem bus AHB (HCLK) dan sistem bus APB (PCLK) serta clock dengan frekuensi lebih rendah, untuk RTC dan timer watchdog. Sumber clock untuk STM32F207 bisa berasal dari internal maupun eksternal. Untuk frekuensi tinggi, sistem clock bisa langsung dari sumber clock tersebut, atau melalui unit PLL sehingga CPU bisa bekerja dengan frekuensi clock lebih tinggi.

Sumber clock internal berasal dari osilator RC internal. STM32F207 dilengkapi dengan 2 buah osilator internal yang dinamakan dengan osilator HSI (*High Speed Internal*) dengan frekuensi 16 MHz dan osilator LSI (*Low Speed Internal*) dengan frekuensi 32 KHz. HSI digunakan untuk clock utama dan LSI digunakan untuk timer watchdog atau pun RTC.

Sumber clock eksternal berasal dari sumber clock eksternal yang dihubungkan ke pin GPIO. Sumber clock ini dinamakan dengan HSE (*High Speed External*) untuk clock utama dan LSE (*Low Speed External*) untuk clock lebih rendah. Pin HSE adalah PH0 untuk masukan osilator (OSC_IN) dan PH1 sebagai keluaran osilator (OSC_OUT), sedangkan pin LSE PC14 untuk masukan (OSC32_IN) dan PC15 untuk keluaran osilator (OSC32_OUT). Sumber clock eksternal bisa berupa osilator eksternal atau pun sebuah kristal atau resonator keramik.

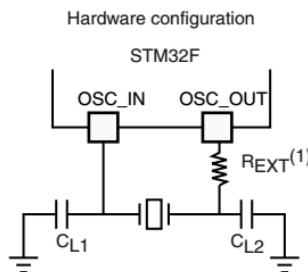
Osilator eksternal dihubungkan ke pin masukan osilator (OSC_IN maupun OSC32_IN). Mode ini dinamakan dengan *HSE Bypass* dan *LSE Bypass*. Sumber sinyal bisa berupa gelombang kotak, sinus maupun sinyal segitiga dengan siklus kerja 50%. Frekuensi untuk HSE 1 – 16 MHz, sedangkan untuk LSE maksimal sampai 1 MHz. Pada kondisi ini, pin keluaran osilator (OSC_OUT dan OSC32_OUT) dibiarkan di kondisi impendansi tinggi.



Gambar 6.7 Sumber Clock dari Osilator Eksternal

Sebuah kristal atau resonator keramik juga bisa dijadikan sebagai sumber clock dari luar. Walaupun pada dasarnya kristal tersebut hanya digunakan untuk menentukan frekuensi osilator internal yang dimiliki oleh STM32F207, tetapi berbeda dengan osilator RC (HSI atau LSI). Kedua kaki kristal dihubungkan dengan kedua pin untuk osilator. Di pin keluaran osilator direkomendasikan untuk menambahkan sebuah resistor (R_{EXT}). Resistor ini berfungsi untuk meredam noise frekuensi tinggi dari osilator dan juga mengurangi konsumsi arus oleh osilator. Nilai R_{EXT} tergantung dengan karakteristik kristal yang digunakan, biasanya

berkisar 220 – 470 Ohm. Kristal juga membutuhkan kapasitor beban agar osilator bisa berfungsi dengan baik. Nilai kapasitor juga ditentukan oleh karakteristik kristal, nilainya berkisar dari 5 – 25 pF untuk HSE dan maksimal 15 pF untuk LSE. R_{EXT} dan kapasitor beban harus diletakan sedekat mungkin dengan pin osilator di PCB.



Gambar 6.8 Sumber Clock dari Kristal

Nilai frekuensi kristal untuk HSE adalah 4 – 26 MHz dan 32,768 KHz untuk LSE. HSE disarankan untuk menggunakan kristal 25 MHz agar bisa menyediakan clock yang akurat untuk ethernet, USB OTG, I2S dan periperal kecepatan tinggi lainnya. Disarankan juga untuk menggunakan LSE dari pada LSI ketika menggunakan RTC, karena LSE bisa menyediakan clock yang lebih akurat dari pada LSI.

4.4 KONFIGURASI BOOT

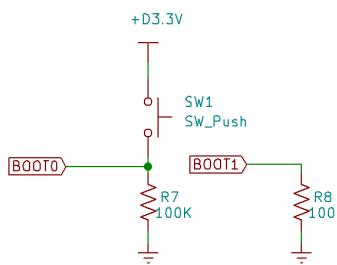
Seperti telah dijelaskan di sub-bab 3.1.4 STM32F207 bisa dikonfigurasi untuk melakukan *booting* (mengambil dan menjalankan program) dari program yang tersimpan di salah satu memori berikut:

1. Memori flash
2. Memori sistem
3. SRAM internal maupun FSMC (SRAM atau flash eksternal)

Pengaturan mode booting ini dilakukan secara hardware melalui 2 buah pin BOOT0 dan BOOT1. Hampir semua aplikasi hanya mengkonfigurasi STM32F207 untuk hanya melakukan booting melalui memori flash, mengingat memori flash 1MB sudah cukup besar. Ada juga yang diatur agar bisa booting dari bootloader agar memudahkan mengganti program di lapangan tanpa perangkat JTAG/SWD. Sedangkan booting melalui SRAM mungkin tidak ada karena SRAM tidak bisa menyimpan program secara permanen. Ada juga aplikasi dari memori flash eksternal (FSMC)

misal apabila STM32F207 digunakan untuk menjalankan sistem operasi Linux (μ C Linux) yang memang membutuhkan memori yang lebih besar.

Untuk booting dari memori flash, pin BOOT0 harus berada di logika rendah saat reset, sedangkan pin BOOT1 diabaikan (tabel 3.2). Oleh karena itu pin BOOT0 biasanya langsung di-pull down ke GND melalui resistor (biasanya 100K). Sedangkan pin BOOT1 digunakan sebagai pin GPIO (PB2). Apabila diinginkan agar memori flash bisa diprogram melalui bootloader, BOOT0 bisa dihubungkan dengan tegangan 3,3V (logika tinggi) melalui sebuah saklar, dan BOOT1 di-pull down melalui resistor. Untuk mengaktifkan bootloadernya, tinggal tekan saklar di BOOT0 diikuti dengan me-reset mikrokontroler.



Gambar 6.9 Konfigurasi Boot untuk Bootloader

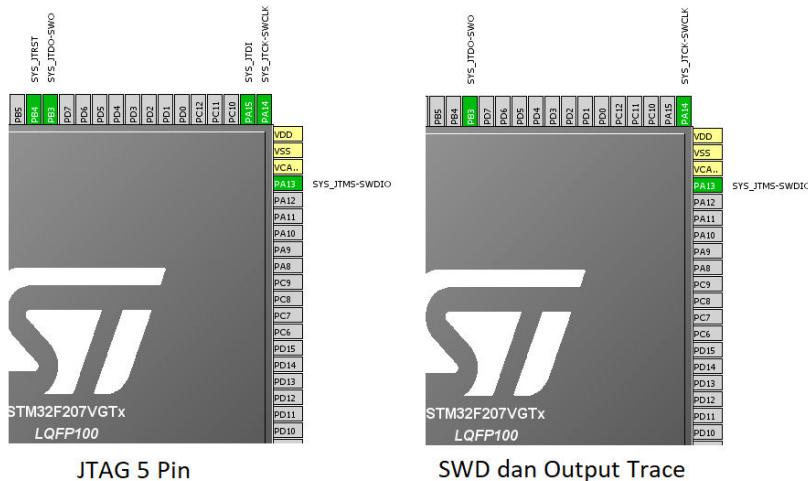
Walapun aplikasi hanya mengijinkan STM32F207 melakukan proses boot dari memori flash, pin BOOT0 sebaiknya tidak langsung dihubungkan ke ground, tetapi tetap melalui resistor pull down. Hal ini untuk menjaga apabila secara sengaja atau tidak, port JTAG/SWD menjadi tidak berfungsi untuk memprogram STM32F207, maka bootloader bisa menjadi penyelamat. Ketika STM32F207 mengalami hal ini, reset STM32F207 ke bootloader, kemudian akses memori flash melalui JTAG/SWD, kemudian hapus isi memori flash. Atau bisa juga isi flash dihapus melalui bootloader. Setelah dihapus, STM32F207 biasanya akan kembali normal.

4.5 SISTEM DEBUG

Ketika mengembangkan aplikasi berbasis mikrokontroler, apalagi aplikasi yang komplek, men-debug program merupakan sebuah keharusan. STM32F207 mempunyai fungsi debug melalui port debug dengan koneksi JTAG/SWD (mengacu kepada sub-bab 2.9 untuk

informasi lebih detil mengenai sistem debug ARM Cortex-M3). Port debug juga digunakan untuk memprogram atau membaca isi memori flash.

Port debug STM32F207 tidak didedikasikan secara khusus, tetapi dipakai bersama dengan fungsi GPIO yang lain. Port debug bisa saja difungsikan sebagai GPIO, namun setelah itu setiap kali reset, port tersebut akan diinisialisasi sebagai GPIO, dan port debugnya kemungkinan besar tidak akan bisa berfungsi. Program tidak bisa di-debug dan untuk sekedar mendownload program ke memori flash pun harus memaksa dulu prosesor masuk ke bootloader, dengan konfigurasi pin boot. Oleh karena itu port debug sebaiknya tidak difungsikan sebagai GPIO.



Gambar 6.10 Debug JTAG dan SWD

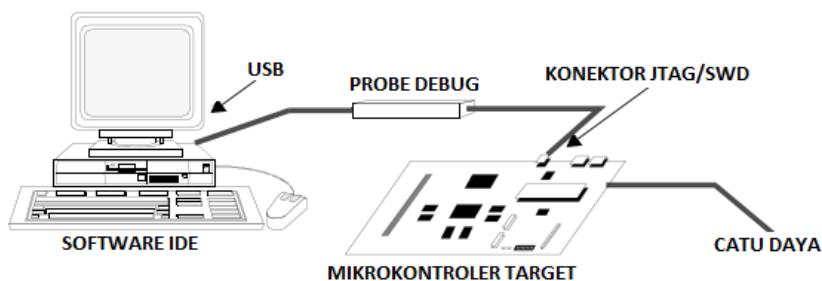
Oleh karena JTAG lebih banyak menggunakan pin, maka SWD menjadi pilihan, apalagi di aplikasi di mana hampir semua pin prosesor dipakai, misal aplikasi dengan FSMC (LCD atau RAM eksternal) atau ethernet dengan antarmuka MII. Port debug biasanya dihubungkan dengan sebuah *male header* atau konektor IDC, atau bisa juga dengan sebuah test point.

by Kang U2Man (u_2man@yahoo.co.id)

Bab 5

ALAT PENGEMBANGAN

Sistem berbasis mikrokontroler (sistem embedded) merupakan sistem hardware dan software. Diperlukan sebuah perangkat pengembangan (*development tool*) berupa hardware dan software. Hardware berupa sebuah probe debug atau adaptor debug yang akan menghubungkan mikrokontroler target dengan software pengembangan yang terinstal di sebuah komputer. Software pengembangan merupakan sebuah IDE (*Integrated Development Environment*) yang biasanya mendukung editor teks untuk membuat program, compiler dan fungsi debug.



Gambar 5.1 Perangkat Pengembangan Mikrokontroler

5.1 PROBE DEBUG

Probe debug akan menghubungkan mikrokontroler target dengan software IDE yang terinstal di PC. Probe debug terhubung dengan mikrokontroler target melalui konektor JTAG/SWD dan terhubung dengan PC melalui USB atau koneksi lain, misal ethernet atau WIFI.



ST LINK (ST Microelectronics)

ULINK2 (Keil)

Gambar 5.2 Probe Debug

ST Microelectronics selaku pabrik STM32F207 membuat probe debug yang dinamakan dengan ST Link, baik yang berupa produk tersendiri maupun yang ada di board evaluasi yang juga dibuatnya. ST Link yang berupa produk tersendiri mendukung koneksi JTAG dan SWD, sedangkan yang ada di board evaluasi biasanya menggunakan koneksi SWD. ST Link hanya mendukung mikrokontroler yang dibuat oleh ST sendiri, keluarga STM32 dan STM8.

Perusahaan perangkat lunak yang membuat IDE dan compiler, Keil (www.keil.com), juga membuat probe debug yang dinamakan dengan ULINK2. Probe debug ini juga mendukung koneksi JTAG dan SWD. ULINK2 mendukung mikrokontroler ARM Cortex-M yang juga didukung oleh IDE yang dibuat oleh Keil.

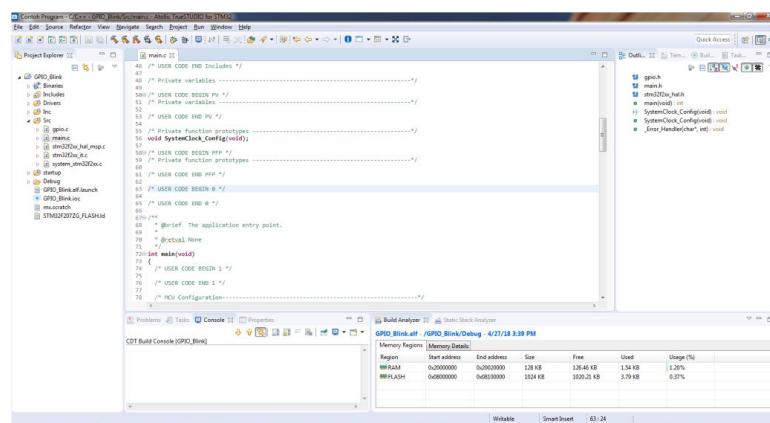
Selain kedua probe debug di atas, banyak juga probe debug yang dibuat oleh pabrikan lain seperti J-Link dari Segger (www.segger.com), bahkan ada probe debug yang bersifat *open source/open hardware*, artinya siapa saja bisa membuat sendiri. Dalam pembahasan buku ini, probe debug menggunakan ST Link.

5.2 IDE

IDE (*Integrated Development Environment*) merupakan software yang digunakan untuk membuat program berbasis mikrokontroler. Disebut *integrated* atau terpadu karena di dalam IDE telah terintegrasi fungsi-fungsi editor teks untuk membuat program, compiler C/C++, assembler, dan linker serta *debugger* untuk mendebug dan mendownload program ke memori flash mikrokontroler target. IDE ada yang berbayar, ada juga yang gratis. Beberapa IDE yang populer untuk pengembangan aplikasi ARM Cortex-M di antaranya TrueStudio dari Atollic, µVision dari Keil,

Embedded Workbench dari IAR, *System Workbench for STM32* dari Ac6 dan masih banyak lagi, misal yang menggunakan Eclipse dan compiler GCC.

TrueStudio merupakan IDE berbasis *Eclipse* yang dibuat oleh Atollic (www.atollic.com). TrueStudio pada awalnya adalah IDE yang berbayar dan mendukung banyak tipe mikrokontroler ARM Cortex-M0/M3/M4/M7 dari berbagai manufaktur silikon (Atmel, TI, ST dan lain-lain). Kemudian ST Microelectronics mengakuisisi Atollic pada bulan Desember 2017 dan menggratiskan TrueStudio, tetapi TrueStudio hanya mendukung mikrokontroler STM32 buatan ST Microelectronics.



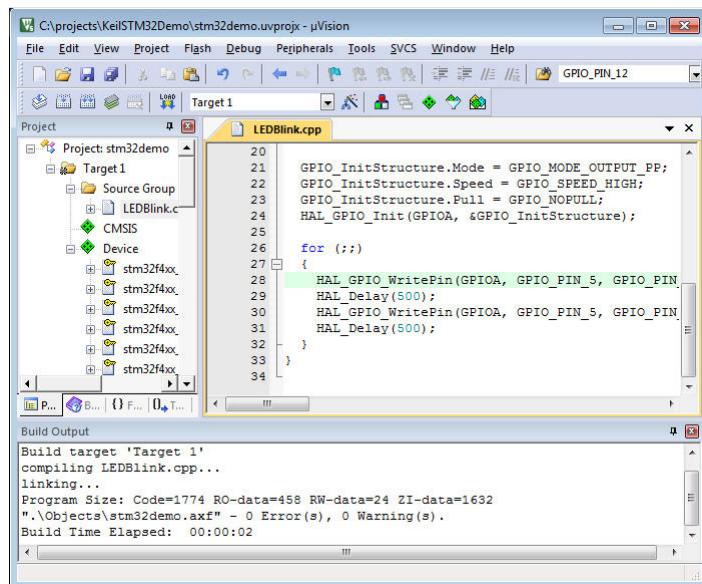
Gambar 5.3 TrueSTUDIO Atollic

TrueStudio yang digratiskan untuk STM32 merupakan TrueStudio dengan fitur-fitur yang ada di versi TrueStudio Profesional (yang berbayar). Fitur-fitur penting yang dimiliki oleh TrueStudio terutama fitur untuk debug, seperti debug untuk RTOS, analisa pemakaian stack dan memori (flash dan RAM), *trace* dan deteksi eksepsi.

IDE µVision merupakan IDE yang menjadi bagian dari core MDK (*Microcontroller Development Kit*) yang dikembangkan oleh Keil (www.keil.com). Keil merupakan perusahaan pertama yang membuat compiler C untuk mikrokontroler 8051. Pada Oktober 2005 Keil diakuisisi oleh ARM Holding.

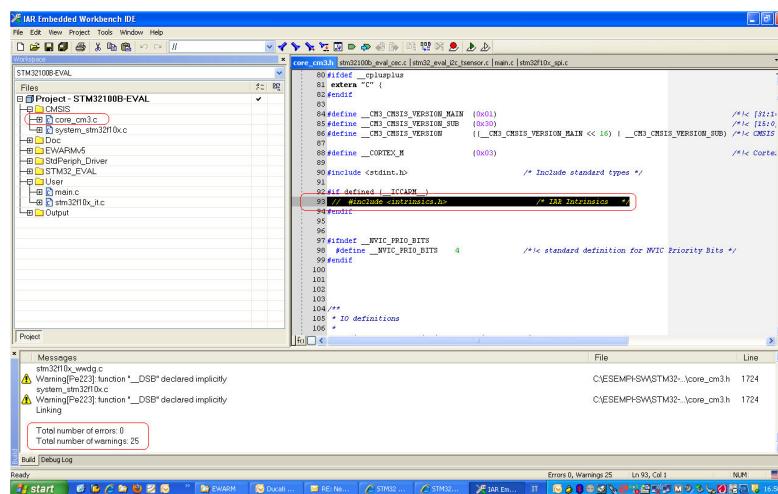
MDK-5 (versi sekarang) selain menawarkan IDE dengan berbagai fitur, juga menawarkan apa yang dinamakan dengan *Software Pack* yang pada dasarnya adalah pustaka (*library*) seperti HAL (*Hardware Abstraction Layer*), sistem file, TCP/IP, USB dan grafis. MDK mendukung tipe mikrokontroler ARM Cortex-M dari berbagai manufaktur silikon. MDK-5

adalah software berbayar walaupun ada juga MDK-Lite yang gratis tetapi ukuran kode yang bisa dibuat dibatasi sampai 32 KB. Ada juga versi MDK yang gratis tetapi khusus untuk ARM Cortex-M0 hasil kerja sama antara Keil dengan manufaktur silikon. Selain ARM Cortex-M, MDK juga mendukung compiler untuk ARM Cortex-A, ARM Cortex-R, ARM7 dan ARM9.



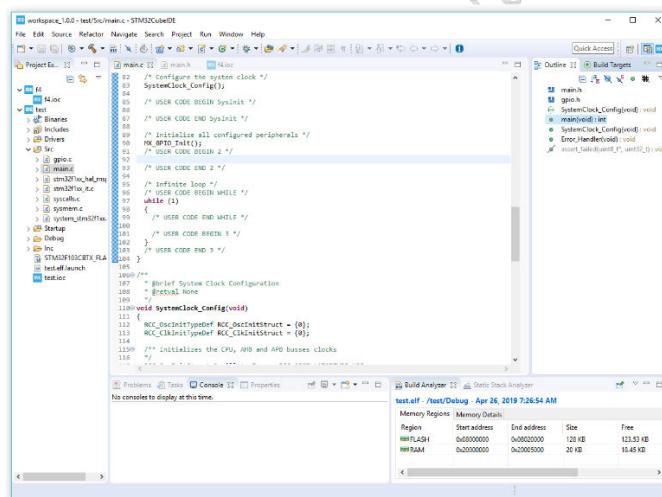
Gambar 5.4 Keil MDK-5

Embedded Workbench for ARM (EWARM) dari IAR (www.iar.com) juga IDE yang mendukung berbagai jenis mikrokontroler ARM Cortex-M, Cortex-R dan juga Cortex-A dari berbagai manufaktur semikonduktor. Dengan fitur yang dinamakan dengan *C-SPY Debugger* yang digunakan untuk analisa performa, pemakaian daya (*power debugging*) dan debug RTOS. EWARM adalah software berbayar, tapi bisa secara gratis selama 30 hari atau dengan pembatasan ukuran program sampai 32 KB.



Gambar 5.5 IAR EWARM

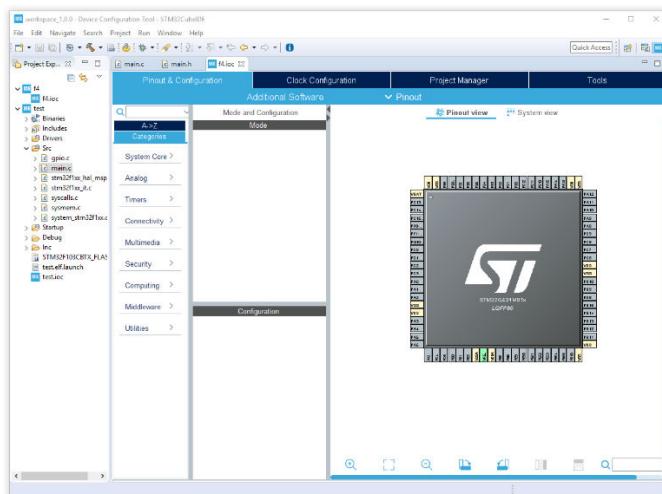
ST Microelectronics juga telah merilis sebuah IDE yang dinamakan dengan STM32CubeIDE. IDE ini pada dasarnya merupakan IDE berbasis Eclipse, yang mempunyai fitur seperti Atollic TrueSTUDIO dan telah diintegrasikan dengan program pendukung STM32CubeMX. STM32CubeIDE bisa diunduh di www.st.com.



Gambar 5.6 STM32CubeIDE

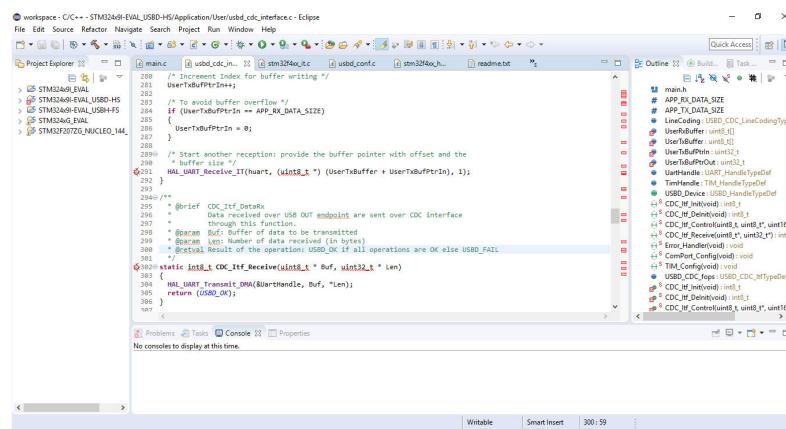
Kelebihan utama STM32CubeIDE adalah telah diintegrasikannya program pendukung untuk pembuatan proyek berbasis STM32 yaitu

STM32CubeMX, sehingga tidak perlu untuk menggunakan 2 program ketika mengembangkan sistem berbasis STM32. Buku ini menggunakan STM32CubeIDE dalam pembuatan contoh program.



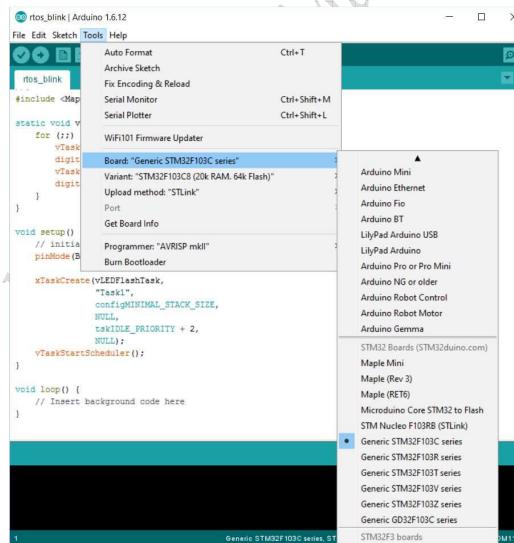
Gambar 5.7 IDE+STM32CubeMX

Selain TrueStudio dan STM32CubeIDE, ada juga IDE gratis untuk mengembangkan program berbasis STM32. IDE yang dinamakan dengan *System Workbench for STM32* merupakan IDE yang berbasis Eclipse dengan compiler GCC. IDE yang bisa diunduh di www.openstm32.org sesuai dengan namanya merupakan IDE khusus STM32. Walaupun fitur debugnya hanya sebatas fitur standar, tapi IDE ini cukup bagus untuk membuat program berbasis STM32.



Gambar 5.8 System Workbench for STM32

Selain IDE yang disebutkan di atas, bisa juga menggunakan Eclipse dengan compiler GCC. Tentu saja harus menambahkan sebuah *plugin* agar bisa dihubungkan dengan probe debug. Ada juga IDE yang dibuat berdasarkan IDE untuk Arduino, sebuah platform mikrokontroler yang awalnya berbasis mikrokontroler AVR dari Atmel (sekarang Microchip). IDE yang dinamakan dengan STM32Duino (www.stm32duino.com) menggunakan bahasa C++ seperti halnya Arduino. Tapi sayangnya IDE ini tidak memiliki fitur debug, walaupun bisa diakali melalui dengan Eclipse dan plugin Arduino.



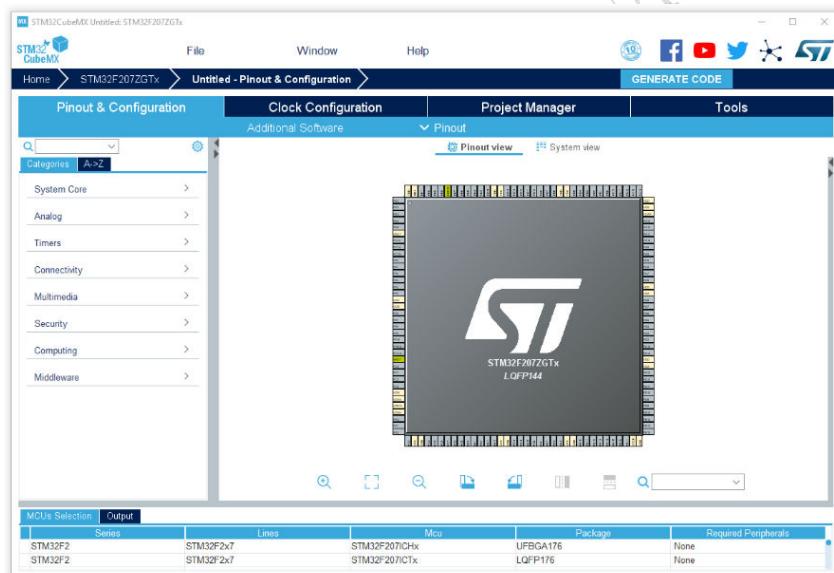
Gambar 5.9 STM32Duino

5.3 SOFTWARE PENDUKUNG

Selain software IDE yang diperlukan untuk pembuatan program, mungkin juga diperlukan software-software yang mendukung pengembangan aplikasi berbasis mikrokontroler, khususnya aplikasi berbasis mikrokontroler STM32F207. Software-software pendukung digunakan untuk membantu membuat program sesuai IDE yang digunakan, atau untuk memprogram memori flash saat proses produksi, untuk melakukan debug melalui port serial atau melihat protokol ethernet atau USB (*sniffer*).

5.3.1 STM32CUBEMX

STM32CubeMX merupakan software yang dikembangkan oleh ST Microelectronics yang berfungsi untuk memudahkan dalam pembuatan program mikrokontroler STM32. STM32CubeMX mengintegrasikan library HAL untuk memudahkan inisialisasi periperal dan pin GPIO. Selain itu STM32CubeMX juga mengintegrasikan *middleware* seperti sistem file, TCP/IP, RTOS, USB dan grafis (untuk LCD).



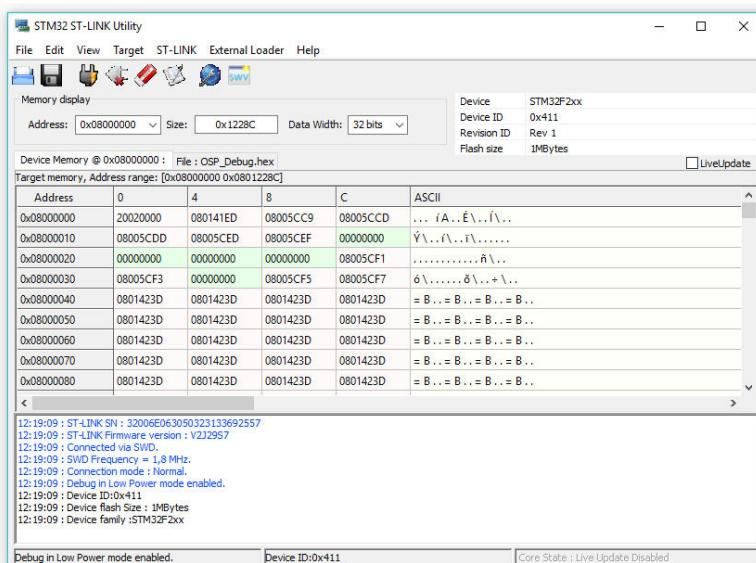
Gambar 5.10 STM32CubeMX

Tentu saja STM32CubeMX tidak bisa melakukan semuanya. STM32CubeMX hanya sebatas menginisialisasi periperal atau middleware

yang digunakan. Apalagi terkadang ditemukan bug di kode yang dihasilkan oleh STM32CubeMX sehingga harus di edit secara manual. STM32CubeMX bisa diunduh di www.st.com. Saat buku ini ditulis, STM32CubeMX sudah dirilis dalam versi 5.2.0.

5.3.2 ALAT PEMROGRAM FLASH

Walaupun semua IDE mempunyai fitur untuk mendownload program hasil kompilasi ke memori flash, karena memang untuk bisa men-debug, program harus didownload dulu ke memori flash, tetapi ada beberapa hal yang tidak bisa dilakukan melalui IDE, misal mengaktifkan fungsi proteksi agar memori flash tidak bisa dibaca lagi setelah diprogram (*readout protection*). Selain itu, tidak efisien juga memprogram flash melalui IDE di saat produksi masal.

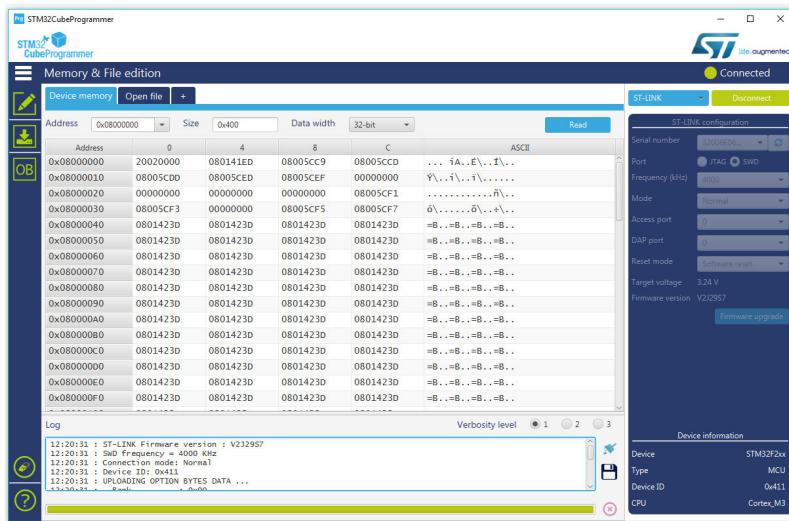


Gambar 5.11 ST Link Utility

ST Microelectronics selaku pembuat STM32 telah menyediakan software yang bisa digunakan untuk memprogram flash memori mikrokontroler STM32 tanpa melalui IDE. Program ini dinamakan dengan *STM32 ST Link Utility*. ST Link Utility ini terkoneksi dengan ST Link dan bisa memprogram melalui JTAG atau SWD. Selain memprogram memori flash, program ini juga bisa digunakan untuk membaca isi flash, melindungi memori flash sehingga tidak bisa dibaca (*read out protection*)

atau diprogram lagi (*write protection*). Selain itu bisa juga digunakan untuk memprogram *option byte* yang dimiliki oleh keluarga STM32.

ST Microelectronics juga mengeluarkan program terbaru yang kemungkinan untuk menggantikan ST Link Utility. Program ini dinamakan dengan *STM32Cube Programmer*. Selain fitur-fitur yang sama dengan ST Link Utility, ST Microelectronics menambahkan fungsi untuk memprogram flash melalui *bootloader* baik melalui UART maupun USB. ST Link Utility dan STM32Cube Progammer juga bisa digunakan untuk meng-update firmware dari ST Link.



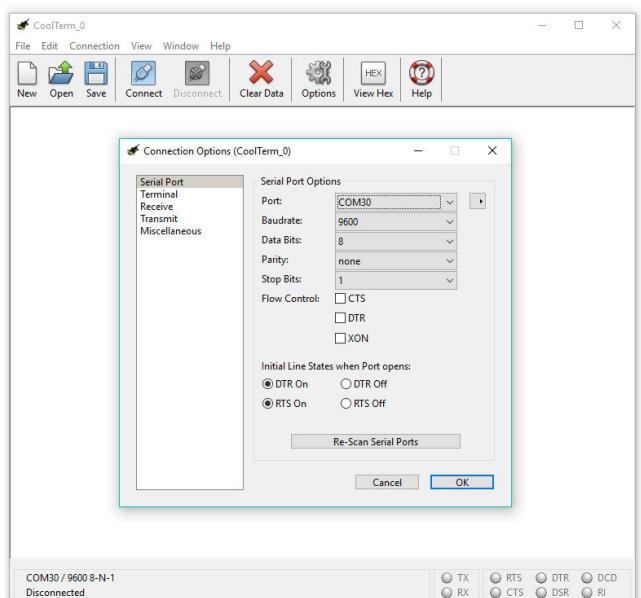
Gambar 5.12 STM32CubeProgrammer

5.3.3 PROGRAM TERMINAL

Hampir sebagian besar aplikasi sistem embedded harus terhubung dengan komputer. Dan sebagian besar antarmukanya menggunakan komunikasi serial (UART), karena lebih mudah di sisi pemrogramannya. Pada saat pengembangan, sebelum aplikasi komputer selesai dibuat, digunakan terlebih dahulu program terminal untuk menguji protokol komunikasi antara mikrokontroler dengan PC. Selain itu, banyak aplikasi mikrokontroler menggunakan port serial sebagai sarana untuk mendebug program, dengan cara mengirimkan status tentang program yang sedang berjalan. Debug melalui port serial juga lebih memudahkan untuk pengecekan perangkat ketika perangkat tersebut sudah terpasang

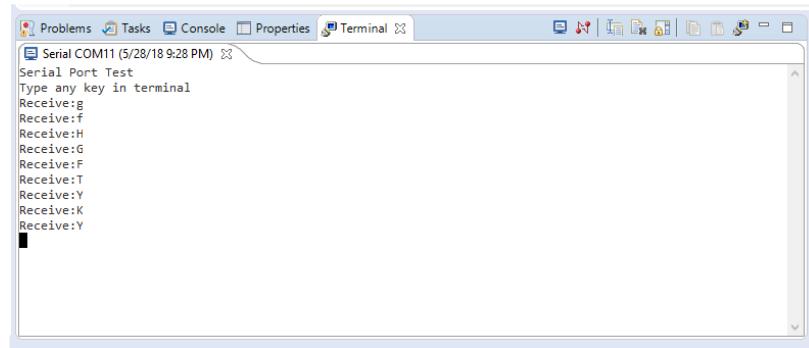
di lapangan karena tidak perlu sambil membuka program melalui IDE dan tidak memerlukan perangkat debug.

Ketika versi Windows XP dan versi sebelumnya, Microsoft menyertakan sebuah program terminal yang sangat populer yang dinamakan *Hyper Terminal*. Tetapi program itu sekarang sudah tidak sertakan lagi ke dalam versi Windows terbaru. Cukup banyak tersedia program terminal yang bisa digunakan secara gratis. Salah satunya adalah *Cool Term* yang bisa diunduh di <http://freeware.the-meiers.org>. *Cool Term* tersedia untuk sistem operasi Windows, MacOS dan Linux. *Cool Term* bisa menerima data dan menampilkannya dalam format ASCII maupun bilangan heksadesimal. Tetapi *Cool Term* hanya bisa mengirim format data ASCII karena pengiriman datanya dilakukan dengan mengetik karakter yang akan dikirim melalui keyboard.



Gambar 5.13 Program Terminal Cool Term

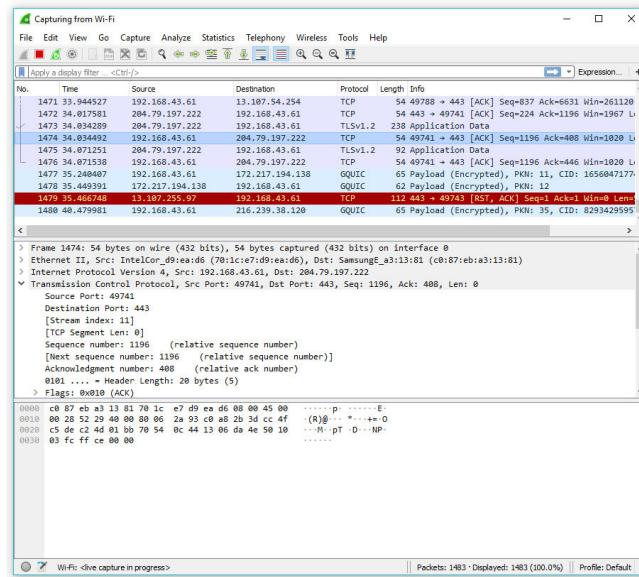
Beberapa IDE seperti TrueSTUDIO juga menyediakan program terminal yang terintegrasi sehingga tidak diperlukan program terminal tambahan.



Gambar 5.14 Program Terminal dari TrueSTUDIO

5.3.4 PROGRAM PENGANALISA PROTOKOL

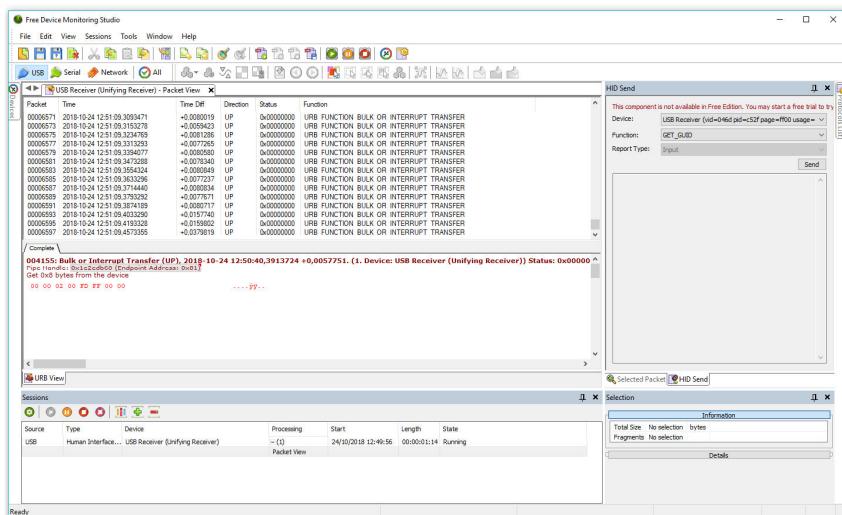
Selain dipergunakan untuk komunikasi dengan PC atau mendebug, program terminal juga bisa digunakan sebagai penganalisa protokol (*protocol analyzer*) untuk komunikasi serial antar 2 perangkat ketika protokol komunikasinya tidak diketahui. Ketika aplikasi menggunakan komunikasi selain serial, misal ethernet atau USB, maka ada baiknya juga menggunakan program yang digunakan untuk menganalisa atau meng-capture lalu lintas data (*data traffic*) di jalur komunikasi.



Gambar 5.15 Wireshark Penganalisa Protokol Jaringan

Salah satu program yang banyak digunakan untuk menganalisa lalu lintas data ethernet atau jaringan TCP/IP (internet) adalah *Wireshark* yang bisa digunakan secara gratis dan dapat diunduh di www.wireshark.org.

Untuk menganalisa protokol USB, bisa digunakan *HHD Software Device Monitoring*, bisa diunduh di <https://freeusbmonitor.com>, walaupun versi gratisnya fungsi-fungsinya dibatasi, tapi cukup untuk sekedar melihat lalu lintas data USB.



Gambar 5.16 Penganalisa Protokol USB

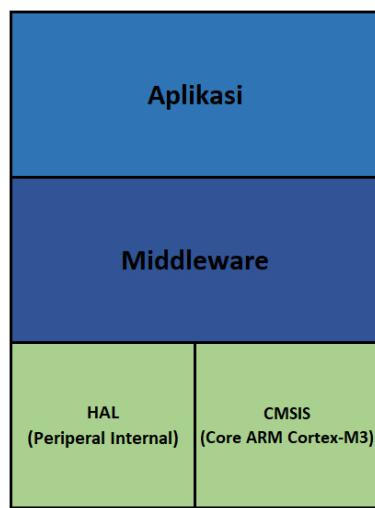
5.4 HAL, CMSIS DAN MIDDLEWARE

Ketika mengembangkan sistem embedded, tentu saja selain hardware ada juga software yang harus dibuat. Software atau sering juga disebut firmware selain membutuhkan perangkat pengembangan seperti yang telah dibahas di sub bab sebelumnya, juga membutuhkan pustaka (library) yang terdiri atas baris-baris kode yang akan membentuk program yang berjalan sesuai dengan sistem yang dibuat.

Sistem embedded pastinya melibatkan hardware dan software yang saling berkomunikasi sesuai dengan fungsi yang telah dirancang. Misal ketika membuat sebuah aplikasi yang akan membaca isi sebuah file dari kartu mikro SD, maka diperlukan program untuk antarmuka dengan mikro SD tersebut, kemudian program untuk menangani sistem file dan program aplikasinya sendiri, artinya akan diapakan file tersebut setelah

bisa dibaca, dikirim lewat port serial atau ditampilkan ke LCD. Program untuk antarmuka ke mikro SD dinamakan dengan HAL (*Hardware Abstraction Layer*), sedangkan program yang menangani sistem file dinamakan dengan *Middleware*.

HAL adalah kumpulan pustaka untuk mengakses periperal internal prosesor, inisialisasi periperal, baca-tulis data dan kendali interupsinya. Setiap pabrikan mikrokontroler pasti telah menyediakan pustaka HAL ini untuk memudahkan para pengguna mikrokontrolernya. Bahkan telah banyak juga yang membuat sebuah software sehingga inisialisasi periperal bisa dilakukan secara visual (GUI), seperti STM32CubeMx, sehingga para pengguna tidak perlu bolak-balik membaca manual yang tebalnya bisa ribuan halaman.



Gambar 5.17 Struktur Firmware Aplikasi Sistem Embedded

Selain HAL yang dibuat oleh prabrikan mikrokontroler yang bersangkutan, ARM selaku pencipta prosesor ARM, juga membuat sebuah kumpulan pustaka yang dinamakan dengan CMSIS (*Cortex Microcontroller Software Interface Standard*). CMSIS dibuat untuk menyederhanakan proses pembuatan firmware aplikasi embedded tanpa tergantung kepada jenis prosesor buatan pabrikan tertentu, karena secara arsitektur memang sama. Salah satu CMSIS yang dikeluarkan oleh ARM adalah CMSIS-Core. CMSIS-Core ini berisi HAL untuk mengakses core prosesor ARM Cortex-M, seperti kendali interupsi (NVIC), timer SysTick,

register SCB, MPU atau FPU (*Floating Point Unit*). CMSIS-Core juga membuat standar untuk penamaan eksepsi sistem.

Sehingga dalam firmware sebuah sistem embedded berbasis prosesor ARM, untuk mengakses periperal yang memang berbeda untuk setiap jenis prosesor, digunakan pustaka HAL yang dikeluarkan oleh pabrikan prosesor tersebut, sedangkan untuk mengakses fungsi CPU digunakan CMSIS-Core yang dikeluarkan oleh ARM.

Sistem embedded juga membutuhkan pustaka untuk melakukan fungsi-fungsi tertentu, misalnya menangani sistem file FAT32, komunikasi TCP/IP atau menangani komunikasi melalui USB atau menangani tampilan di LCD (*GUI, Graphic User Interface*). Pustaka ini biasa dinamakan dengan *middleware*. Middleware bisa dibuat oleh pabrikan mikrokontroler atau pihak ketiga yang bisa berbayar maupun digratiskan (bersifat open source). Bahkan ARM juga membuat CMSIS untuk middleware, misal CMSIS-RTOS (*Real Time Operating System*) dan CMSIS-DSP (*Digital Signal Processing*).

ST Microelectronics sudah mengintegrasikan beberapa middleware yang bersifat open source ke program STM32CubeMX untuk memudahkan para programer sistem embedded. Untuk menangani sistem file FAT32, STM32CubeMX menggunakan FATFS dari www.elm-chan.org. Komunikasi ethernet (TCP/IP) menggunakan LwIP (*Light weight IP*) sedangkan untuk komunikasi USB, STM32CubeMX menggunakan middleware yang dibuat oleh ST sendiri. Untuk RTOS digunakan CMSIS-RTOS dari ARM dan FreeRTOS (www.freertos.org).

Pembuat prosesor sering kali juga bekerja sama dengan perusahaan pembuat middleware, sehingga sebuah middleware yang tadinya berbayar bisa digratiskan khusus untuk prosesor tersebut. Misalnya ST Microelectronics bekerja sama dengan Segger (www.segger.com) yang membuat middleware untuk GUI yaitu emWin. Kerja sama tersebut menghasilkan middleware baru yang dinamakan dengan STemWin dan middleware yang tadinya harganya ribuan Euro menjadi gratis. Tetapi ST mendistribusikannya dalam bentuk library yang telah dicompile, tidak disertakan kode programnya. Dan hanya bisa dipakai di mikrokontroler STM32.

Bagian terakhir dari struktur firmware sistem embedded adalah aplikasi. Karena aplikasi bersifat khusus, maka aplikasi harus dibuat dari nol.

Tugas utama seorang programer sistem embedded adalah membuat aplikasi dan mengintegrasikannya dengan HAL dan middleware.

Bab 6

BERMAIN-MAIN DENGAN PERIPERAL DASAR

Akhirnya setelah 5 bab sebelumnya membahas tentang teori yang berkaitan dengan mikrokontroler ARM Cortex-M3 dan STM32F207, dimulai dari bab ini dan seterusnya akan dibahas tentang contoh-contoh proyek dengan menggunakan mikrokontroler STM32F207. Pembahasan dimulai dengan penggunaan periperal dasar yang dimiliki oleh STM32F207 (misal GPIO, Timer, UART) dan dilanjutkan dengan fitur-fitur canggih yang dimiliki oleh STM32F207ZG, seperti ethernet, USB, SDIO (μ SD), antarmuka FSMC dengan LCD TFT, dan lain-lain.

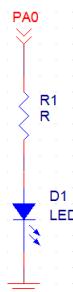
Dalam pembuatan contoh program, digunakan STM32CubeIDE dari ST Micro sebagai IDE yang sudah digabung dengan STM32CubeMx untuk pembuatan proyek (program) dari ST Microelectronics. Pada saat buku ini ditulis, STM32CubeIDE menggunakan versi 1.0.1. Versi terbaru STM32CubeIDE bisa diunduh di www.st.com.

6.1 BERMAIN-MAIN DENGAN GPIO

Sebuah GPIO bisa difungsikan sebagai masukan (input) maupun sebagai keluaran (output). Sebagai input sebuah GPIO bisa digunakan untuk membaca sebuah saklar, keluaran opto coupler atau sinyal digital lainnya, yang tentu saja level digitalnya harus sesuai dengan level digital GPIO STM32F207 yang normalnya di tegangan 3,3 Volt walaupun bisa juga dihubungkan langsung dengan level logika 5 Volt karena GPIO STM32 bersifat 5V tolerant. Sebagai output GPIO bisa digunakan untuk menyalakan LED, mengontrol relay, membunyikan buzzer dan lain-lain.

6.1.1 GPIO SEBAGAI OUTPUT

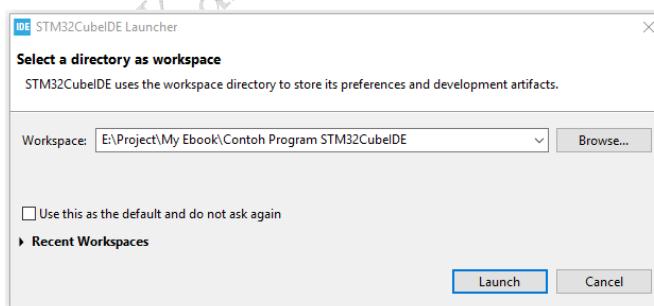
Pada proyek pertama ini, GPIO akan digunakan untuk mengendalikan sebuah LED yang terhubung melalui sebuah resistor ke GPIO yaitu GPIO Port A pin ke-3 atau PA3. Rangkaian atau sirkuit untuk mengendalikan LED ini ditunjukkan oleh gambar 6.1. Dengan rangkaian seperti ini, diperlukan logika tinggi (set) di PA0 untuk menyalakan LED, dan logika rendah (reset) untuk mematikannya.



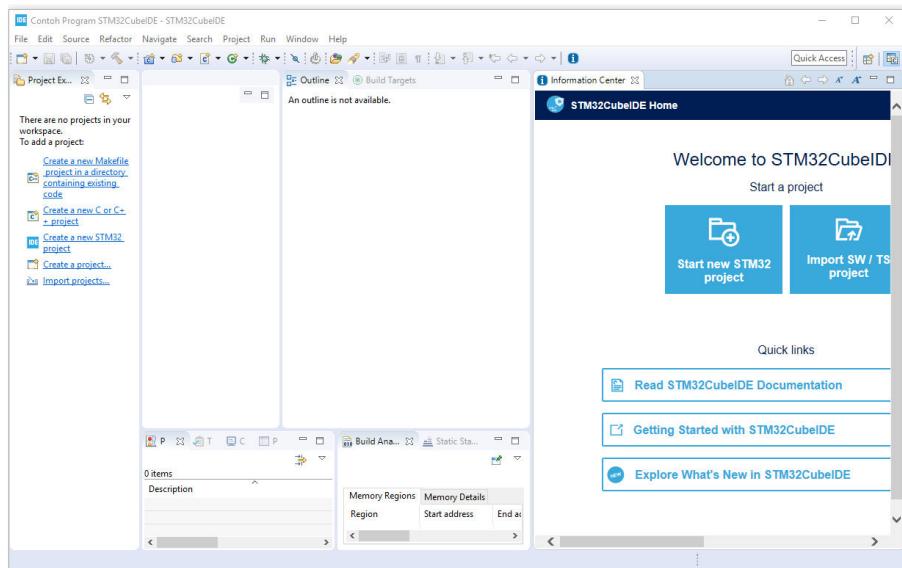
Gambar 6.11 Rangkaian Pengendali LED

6.1.1.1 Pembuatan Proyek Dengan STM32CubeIDE

Ketika STM32CubeIDE dijalankan, program akan meminta untuk memilih direktori/folder sebagai workspace. Setelah tampilan utama STM32CubeIDE keluar, apabila lokasi workspace tersebut pertama kali dibuka atau belum ada proyek yang dibuat, di *Project Explorer* tidak akan terdapat daftar proyek. Untuk itu proyek bisa dibuat baru atau di-impor dari folder lain. Untuk membuat proyek baru bisa menggunakan menu *Create a new STM32 project* dari *Project Explorer* atau tombol *Start new STM32 project* dari jendela *Information Center*



Gambar 6.12 Pemilihan Workspace STM32CubeIDE



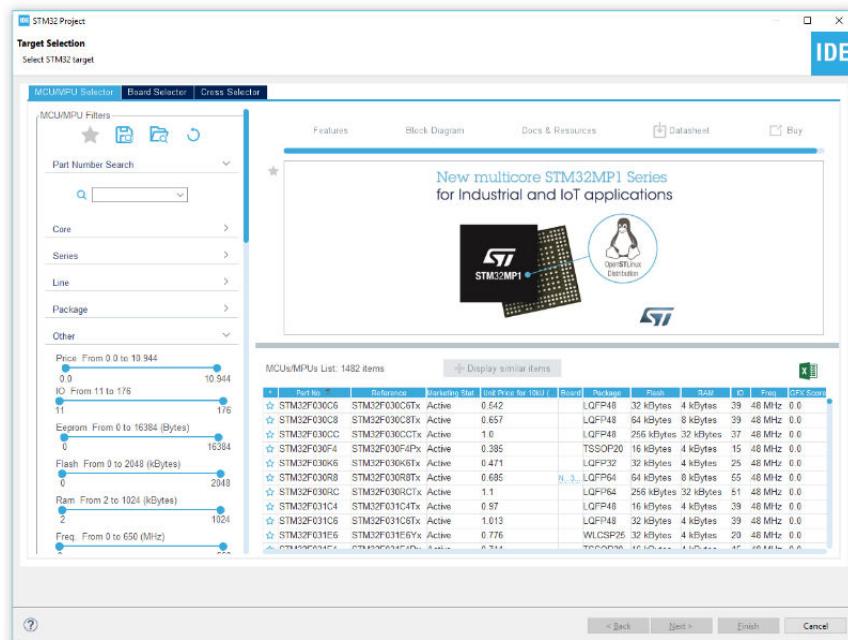
Gambar 6.13 Tampilan Utama STM32CubeIDE

IDE akan menginisialisasi STM32CubeMX yang sudah terintegrasi. Proses inisialisasi mungkin akan berlangsung agak lama, karena IDE akan memperbaharui database yang dimilikinya (daftar tipe STM32) dan mengecek juga kalau ada versi terbaru dari IDE dengan mengunduh dari server yang dimiliki oleh ST. Proses ini bisa dihentikan. Kemudian sebuah jendela baru dengan 3 tab akan muncul.

1. MCU/MPU Selector
2. Board Selector
3. Cross Selector

Proyek bisa dibuat dengan memilih tipe MCU, board evaluasi atau *Cross Selector* yang akan membandingkan MCU dari pabrikan lain dan persamaannya dengan STM32. Pemilihan MCU bisa dilakukan berdasarkan Core (Cortex-M0, M3, M4, M7 dan lain-lain), atau berdasarkan seri (F0,F1, F2, F3, F4, F7). Bisa juga memilih MCU berdasarkan kemasan yang akan digunakan, atau memilih berdasarkan kisaran harga dan periperal yang digunakan. Setiap kali kriteria dipilih, STM32CubeMX akan menampilkan jenis STM32 yang sesuai dengan kriteria tersebut di daftar MCU/MPU. Untuk core M4 dan M7 bisa mengaktifkan middleware *Advanced Graphic* dan *Artificial Intelligence (AI)*. Untuk core yang lain, seperti STM32F207 yang memiliki core Cortex-M3,

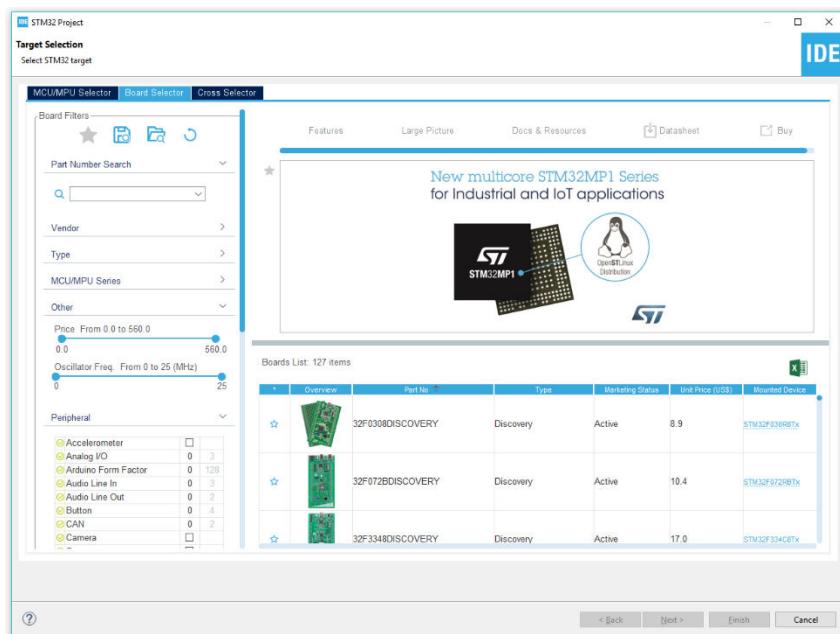
middleware tersebut masih bisa digunakan dengan cara ditambahkan secara manual.



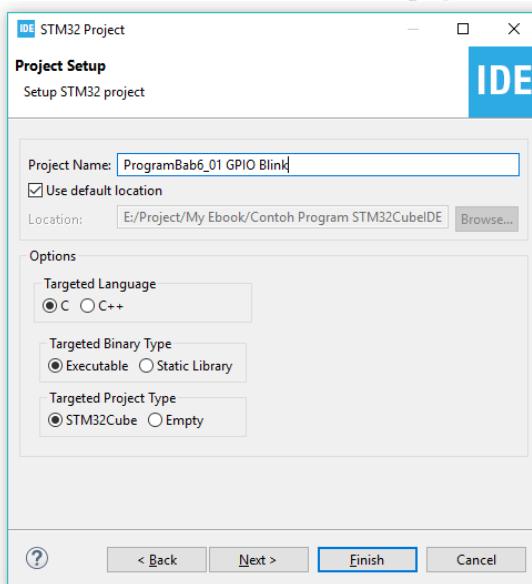
Gambar 6.14 Pembuatan Proyek Baru Berdasarkan Tipe STM32

Jika diinginkan pembuatan proyek berdasarkan board pengembangan buatan ST Microelectronics bisa dilakukan dari tab *Board Selector*. Pemilihan berdasarkan board bisa memilih antara board *Discovery*, *Nucleo* atau *Evaluation Board*. Discovery dan Nucleo merupakan board evaluasi dengan fitur minimal, hanya mengeluarkan hampir semua pin MCU melalui sebuah header. Sedangkan board Evaluation (EVAL) merupakan board evaluasi yang hampir memfungsikan semua fitur yang dimiliki oleh STM32 yang dipakainya, misal USB, ethernet, FSMC (LCD dan SRAM eksternal) dan lain-lain.

Setelah memilih tipe STM32 yang akan digunakan, dalam hal ini STM32F207ZG dan meng-klik tombol *Next*, STM32CubeIDE akan meminta untuk memasukan nama proyek yang akan dibuat. Misal nama proyek yang dibuat *ProgramBab6_01 GPIO Blink*.

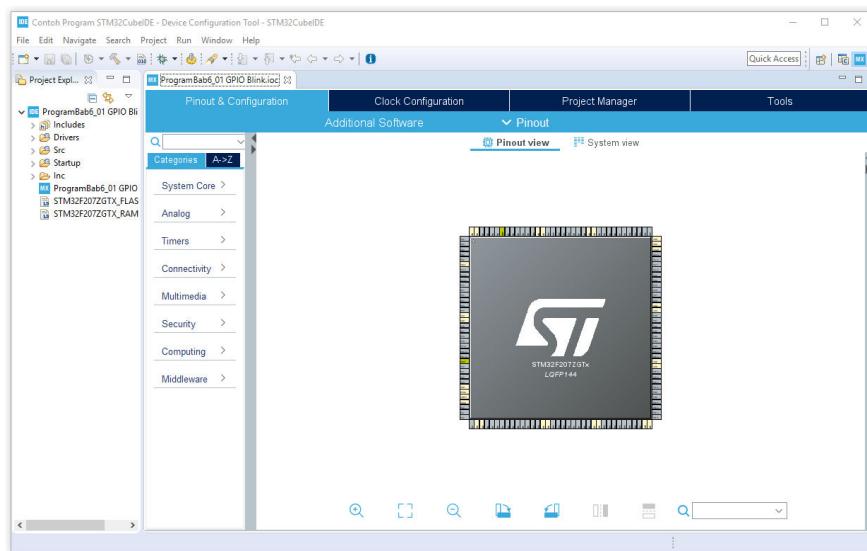


Gambar 6.15 Pemilihan Berdasarkan Board Pengembangan



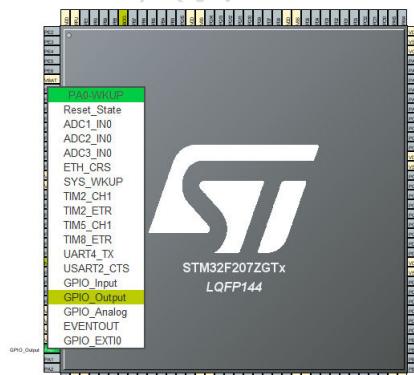
Gambar 6.16 Pemberian Nama Projek

Setelah mengklik *Finish* IDE akan membuat proyek kemudian menampilkan pin-out dari tipe MCU yang digunakan.



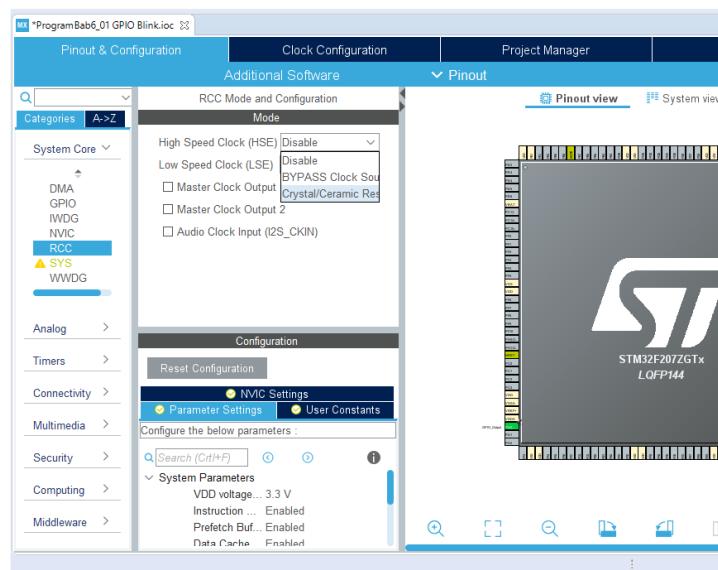
Gambar 6.17 Pinout dan Periperal STM32F207ZG

Untuk mengaktifkan sebuah pin GPIO, tinggal klik di GPIO tersebut, misal dalam contoh program yang akan dibuat adalah PA0. STM32CubeMx kemudian akan menampilkan sebuah menu *pop up* yang menampilkan semua fungsi alternatif dari pin tersebut. Untuk memfungksikan sebagai output pilih *GPIO_Output*.



Gambar 6.18 Pengaturan Pin GPIO Sebagai Output

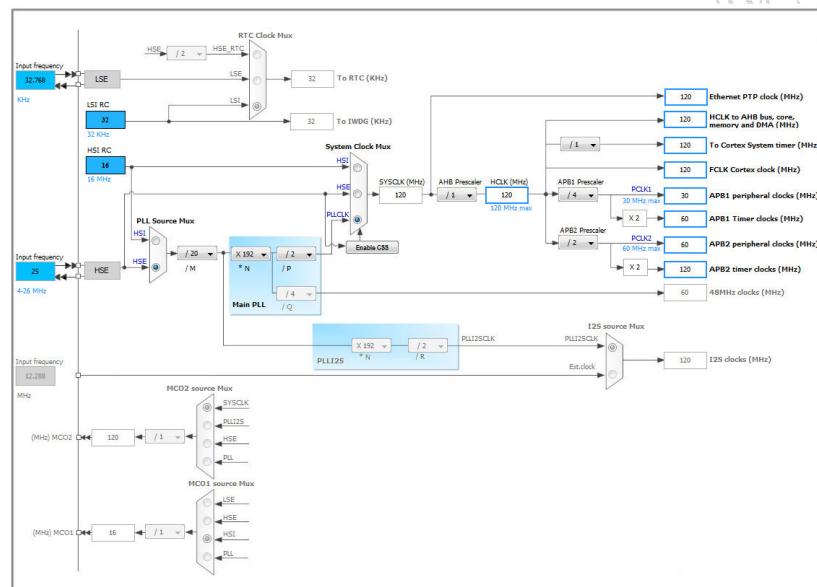
Langkah berikutnya adalah pengaturan clock. Seperti telah dijelaskan di bab sebelumnya, STM32 bisa bekerja dengan sumber clock eksternal maupun internal, baik itu untuk clock utama (frekuensi tinggi) maupun clock sekunder (frekuensi rendah). Untuk memilih sumber clock, klik *System Core* kemudian pilih *RCC*. Dari jendela *Pinout* pilih HSE untuk clock frekuensi tinggi dan LSE untuk frekuensi rendah. Ada 3 pilihan: *Disable*, *BYPASS Clock Source*, dan *Crystal/Ceramic Resonator*. *Disable* artinya sumber clock akan diambil dari osilator internal (HSI dan LSI), *BYPASS Clock Source* artinya sumber clock berasal dari sebuah osilator eksternal melalui pin *OSC_IN* (pin *PH0*) untuk HSE dan pin *OSC32_IN* (pin *PC14*) untuk LSE. Sedangkan *Crystal/Ceramic Resonator* artinya sumber berasal dari sebuah kristal atau resonator keramik.



Gambar 6.19 Pengaturan Sumber Clock

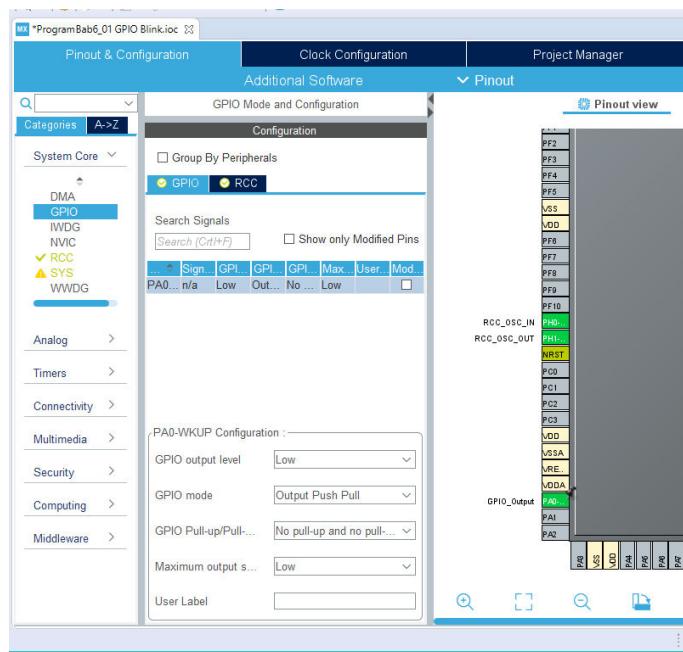
Setelah menentukan sumber clock, langkah selanjutnya adalah melakukan pengaturan clock melalui tab *Clock Configuration*. STM32 mempunyai kebutuhan clock yang berbeda-beda. SYCLK adalah clock yang diperlukan oleh core Cortex-M3, HCLK adalah clock yang diperlukan untuk bus AHB dan PCLK clock untuk bus APB (periperal). Oleh karena setiap bus membutuhkan kecepatan clock yang berbeda, maka terdapat sebuah pembagi clock (*prescalar*) agar kecepatan clock diatur sesuai dengan spesifikasinya, 120 MHz untuk bus AHB, 30 MHz untuk APB1 dan 60 MHz untuk APB2.

Ketika sumber clock berasal dari luar, nilai frekuensinya harus dimasukan ke kotak *Input frequency* (misal 25 MHz). Sumber clock sesungguhnya ditentukan oleh *System Clock Mux* apakah berasal dari internal (HSI), eksternal (HSE) atau dari unit PLL (PLCLK). Untuk mendapatkan kecepatan clock yang tinggi maka sumber clock perlu dipilih dari PLCLK. Unit PLL sendiri mempunyai masukan yang bisa dipilih dari HSI atau HSE (*PLL Source Mux*). Unit PLL mempunyai 2 buah pembagi (M dan P) dan sebuah pengali (N). Ketika nilai frekuensi melebihi nilai maksimum, kotak-kotak frekuensi tersebut akan berwarna merah. STM32CubeMx bisa melakukan perhitungan otomatis untuk pengaturan clock ini. Setelah memilih sumber sistem clock, yaitu PLCLK, dan sumber PLL, yaitu HSE, tinggal memasukan nilai clock yang diinginkan di kotak HCLK, dalam hal ini 120 MHz. Setelah menekan tombol Enter, STM32CubeMx akan mengatur clock secara otomatis sesuai spesifikasi STM32F207.



Gambar 6.20 Pengaturan Clock

Setelah sumber clock diatur dengan benar, langkah selanjutnya adalah mengatur periperal atau middleware, kembali ke tab *Pinout & Configuration* dan dari System Core, pilih GPIO. Dalam contoh program yang akan dibuat, tidak menggunakan middleware dan hanya GPIO yang diaktifkan.



Gambar 6.21 Konfigurasi Middleware dan Periperal

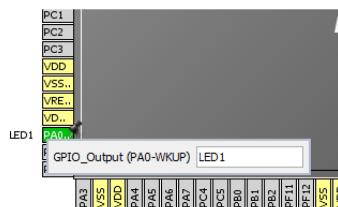
Ada berapa hal yang bisa diatur untuk GPIO. *GPIO output level* menentukan nilai keluaran dari pin GPIO tersebut saat pertama kali program dijalankan, *Low* atau *High*. Agar LED awalnya mati saat pertama kali program jalan, keluaran GPIO harus berlogika rendah (*Low*). *GPIO Mode* mengatur mode output dari GPIO (*Output Push Pull* atau *Open Drain*). Untuk mengendalikan LED, GPIO harus bekerja di mode push pull. *GPIO Pull-up/Pull-down* akan mengatur apakah GPIO tersebut akan dihubungkan dengan resistor pull-up (terhubung ke *VDD*) atau pull down (terhubung ke *GND*) internal atau tidak di pull-up atau pull-down sama sekali. Untuk LED tidak diperlukan pull-up atau pull-down. *Maximum Output Speed* menentukan kecepatan dari GPIO tersebut. Dalam pengendalian LED tidak diperlukan kecepatan tinggi.

Menu terakhir adalah *User Label*. Menu digunakan untuk menamakan pin GPIO yang digunakan. Pustaka HAL telah menentukan nama untuk setiap GPIO (sabagai port dan pin). Misal untuk PA0, HAL telah memberikan nama *GPIO_Port_GPIOA* sebagai nama portnya dan *GPIO_PIN_0* sebagai nama pinnya. User Label digunakan untuk memudahkan dalam pemrograman dengan memberikan nama yang

mudah diingat ke pin tersebut misal LED1. Ketika sebuah pin diberikan label, STM32CubeIDE akan membuat sebuah definisi di dalam program, misal untuk LED1

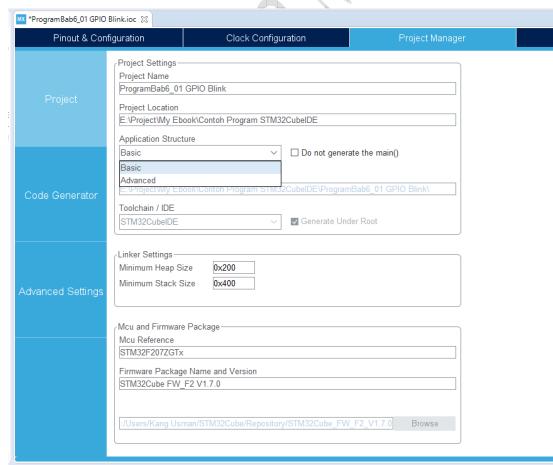
```
#define LED1_Pin GPIO_PIN_0
#define LED1_GPIO_Port GPIOA
```

Pemberian label juga bisa dilakukan dari tab Pinout. Klik kanan di pin yang akan diberikan label. Ketika muncul menu pop up, pilih *Enter User Label* kemudian isikan label di kotak masukan.



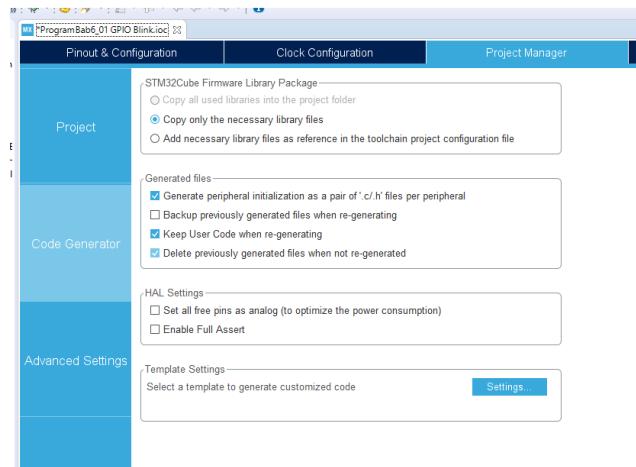
Gambar 6.22 Pemberian Label dari Pinout

Dari sini, pengaturan periperal sudah selesai. Selanjutnya pengaturan untuk proyek program sendiri melalui tab *Project Manager*. Dari menu *Project*, akan muncul jendela untuk mengatur proyek program yang akan dibuat. Nama proyek dan lokasi penyimpanan sudah diatur saat pembuatan proyek. Apabila diinginkan bisa dipilih untuk *Application Structure* yang akan mengatur format folder dan ukuran memori *Heap* dan *Stack*.



Gambar 6.23 Pengaturan Proyek

Klik di tab *Code Generator*. Di bagian *Generated File*, cek *Generate peripheral initialization as a pair of '.c/.h' files per peripheral* agar STM32CubeIDE menghasilkan file source code (file .c dan file .h) untuk masing-masing periperal yang diaktifkan, misal untuk proyek ini akan file periperal yang dihasilkan adalah gpio.c dan gpio.h. Kemudian simpan agar IDE men-generate code.



Gambar 6.24 Menu Code Generator

6.1.1.2 Pembuatan Program

Setelah proyek berhasil dibuat, program segera bisa dibuat. Perlu diingat STM32CubeIDE tidak melakukan semuanya, karena STM32CubeIDE melalui STM32CubeMx-nya hanya menginisialisasi periperal dan middleware yang dipakai oleh proyek yang dibuat. Fungsi-fungsi agar program sesuai dengan yang diinginkan tetap harus dibuat sendiri. Dari Project Explorer dan folder src, klik ganda file main.c untuk membukanya.

Listing Program 6.1 Fungsi Main

```
int main(void)
{
    /* USER CODE BEGIN 1 */

    /* USER CODE END 1 */

    /* MCU Configuration-----*/
}
```

```

/* Reset of all peripherals, Initializes the Flash interface and
the Systick. */
HAL_Init();

/* USER CODE BEGIN Init */

/* USER CODE END Init */

/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
/* USER CODE BEGIN 2 */

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

}
/* USER CODE END 3 */
}

```

Listing program 6.1 adalah potongan program yang diambil dari file main.c yang dihasilkan oleh STM32CubeIDE yang sebelumnya dibuat. Dalam bahasa C, program akan selalu diawali dengan fungsi main. Tetapi setelah reset, mikrokontroler akan mengerjakan sebuah program *startup* (file startup_stm32f207xx.s) untuk inisialisasi awal, seperti mengatur memori *stack* dan *heap*, mengatur vektor reset, dan memanggil fungsi main. Besarnya memori stack dan heap bisa diatur saat pembuatan proyek STM32CubeIDE dari tab Project Setting dan disimpan file sebuah file *linker script* (STM32F207ZG_FLASH.ld). Pada saat startup jalan, STM32F207 akan bekerja dengan osilator internal (16 MHz).

Program main selalu diawali dengan memanggil fungsi HAL_Init() yang akan menginisialisasi library HAL, mengatur agar interupsi timer SysTick

setiap 1 milidetik dan mengatur prioritas group NVIC dengan nilai 4. Selanjutnya akan memanggil fungsi `SystemClock_Config()`, yang terutama akan mengatur sistem clock seperti yang ditentukan oleh STM32CubeMx (bekerja dengan kristal 25 MHz dan frekuensi 120MHz dari PLL). Terakhir program akan memanggil fungsi-fungsi untuk menginisialisasi periperal yang digunakan dalam contoh program ini adalah GPIO.

Listing Program 6.2 Inisialisasi GPIO

```
void MX_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct;

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOH_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_RESET);

    /*Configure GPIO pin : PtPin */
    GPIO_InitStruct.Pin = LED1_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(LED1_GPIO_Port, &GPIO_InitStruct);
}
```

Pada contoh program pertama ini, hanya akan membuat sebuah LED yang terhubung ke PA0 nyala-mati secara bergantian (*blink*). Program bisa ditempatkan di *loop* main dengan memanggil fungsi-fungsi GPIO sebagai output. Bisa menggunakan fungsi `HAL_GPIO_Write_Pin` atau `HAL_GPIO_TogglePin`.

`HAL_GPIO_WritePin`, untuk menset atau me-reset pin GPIO. Fungsi ini memerlukan 3 parameter yaitu port GPIO, pin GPIO, dan status pin. Status pin bisa set (`GPIO_PIN_SET`) atau reset (`GPIO_PIN_RESET`).

Listing Program 6.3 Fungsi GPIO Write

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */
```

```

/* USER CODE BEGIN 3 */
//LED ON
HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_SET);
HAL_Delay(500);
//LED OFF
HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_RESET);
HAL_Delay(500);
}
/* USER CODE END 3 */

```

HAL_GPIO_TogglePin, untuk men-*toggle* pin GPIO, artinya nilai yang akan dikeluarkan di pin tersebut berkebalikan dari nilai pin sebelumnya, kalau pin sebelumnya di-set, maka perintah toggle akan me-reset pin tersebut.

Listing Program 6.4 Fungsi GPIO Toggle

```

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
    HAL_Delay(500);
}
/* USER CODE END 3 */

```

Agar LED on/off, diperlukan jeda (*delay*) di antara LED on dan LED off. HAL telah menyediakan fungsi delay (*HAL_Delay*) dalam milidetik. Fungsi ini menggunakan timer SysTick yang diprogram untuk interupsi setiap 1 milidetik.

Listing Program 6.5 Inisialisasi Timer SysTick

```

/**Configure the Systick interrupt time
*/
HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

/**Configure the Systick
*/
HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

/* SysTick_IRQn interrupt configuration */
HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);

```

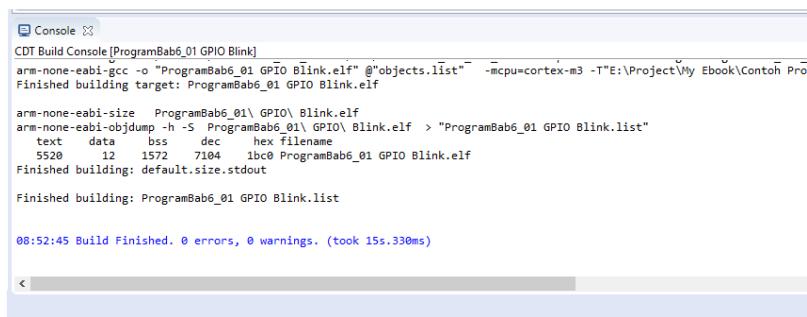
HAL tidak menyediakan fungsi untuk menulis ke GPIO sebagai port (16 bit). Jika diperlukan untuk menulis ke sebuah port secara paralel, maka harus dilakukan dengan mengakses register GPIO secara langsung.

Ada aturan ketika memodifikasi program yang dihasilkan oleh STM32CubeMx. Setiap kali proyek STM32CubeMx men-generate ulang code, akan menimpa (*overwrite*) file-file yang dihasilkannya. Oleh karena itu, STM32CubeMx menyediakan tempat agar program yang sudah diketik tidak hilang saat *generate code* ulang. STM32CubeMx menyediakan baris yang diawali dengan /* USER CODE BEGIN xxx*/ dan diakhiri dengan /* USER CODE END xxx*/. Xxx bisa angka kalau berada di baris kode, *include* untuk menambahkan file header di deklarasi *include*, PV (*private variable*) untuk deklarasi variable atau bisa juga PFP (*private function prototype*) untuk deklarasi fungsi. Pastikan juga menu *Keep User Code when re-generating* di menu Project Manager/Code Generator.

6.1.1.3 Compile, Download dan Debugging

Sebelum bisa di-download ke memori flash mikrokontroler, program harus di-compile terlebih dahulu. Compiler akan mengecek apakah program yang ditulis sesuai dengan kaidah atau aturan penulisan sintak bahasa C. Proses compile pada dasarnya menerjemahkan program bahasa C ke dalam bahasa assembly mikrokontroler (ARM Cortex-M3). Ketika tidak ada kesalahan sintak, compiler kemudian akan melakukan *link* untuk membuat file yang bisa di-download (*executable*) yang bisa berupa file .elf (*executable and linkable format*), file .hex atau binary (.bin). File .elf biasa dipakai oleh IDE saat debug program sedangkan .hex dan .bin biasanya digunakan untuk keperluan produksi.

Compile dilakukan dengan meng-klik menu *Project/Build Project* atau icon bergambar palu. Proses compile bisa diamati di jendela *console*. STM32CubeIDE akan menginformasikan jika terdapat error di program. Baris kode yang error bisa dilihat di jendela *Problem*. Dengan mengklik ganda di jendela *Problem*, STM32CubeIDE akan langsung mengarahkan kurSOR ke baris kode yang error tersebut. Jika tidak ada error, STM32CubeIDE akan membuat file .elf.



```

CDT Build Console [ProgramBab6_01 GPIO Blink]
arm-none-eabi-gcc -o "ProgramBab6_01 GPIO\Blink.elf" @"objects.list" -mcpu=cortex-m3 -T"E:\Project\My Ebook\Contoh Pro
Finished building target: ProgramBab6_01 GPIO Blink.elf

arm-none-eabi-size ProgramBab6_01\GPIO\Blink.elf
arm-none-eabi-objdump -h -S ProgramBab6_01\GPIO\Blink.elf > "ProgramBab6_01 GPIO Blink.list"
      text    data     bss     dec   filename
 5520      12   1572    7104  1bc0 ProgramBab6_01 GPIO Blink.elf
Finished building: default.size.stdout

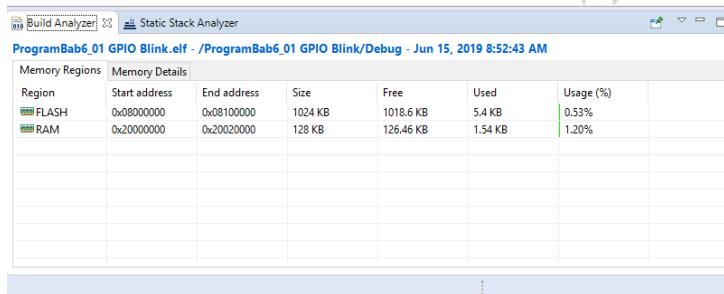
Finished building: ProgramBab6_01 GPIO Blink.list

08:52:45 Build Finished. 0 errors, 0 warnings. (took 15s.330ms)

```

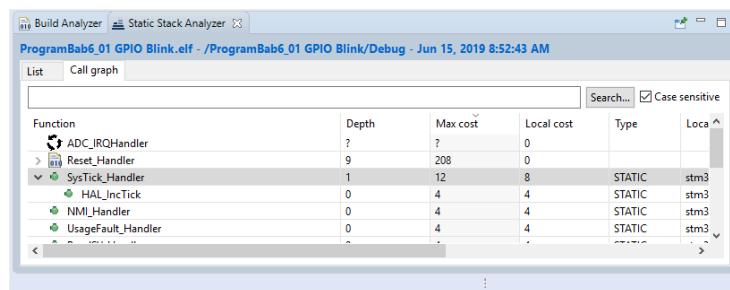
Gambar 6.25 Jendela Console

Dari file .elf ini, STM32CubeIDE akan melakukan analisa terhadap pemakaian memori (flash dan RAM) dan akan menampilkannya di jendela *Build Analyzer*. Tab *Memory Region* akan memberikan pemakaian memori flash dan RAM sedangkan tab *Memory Details* akan memberikan pemakaian secara rinci dari memori tersebut seperti yang terdapat dalam file linker script (.ld). Jika jendela ini belum tampil, Build Analyzer bisa ditampilkan melalui menu *Window>Show View/Other* lalu pilih Build Analyzer. Begitu juga kalau ingin menampilkan jendela-jendela yang lain.



Gambar 6.26 Build Analyzer

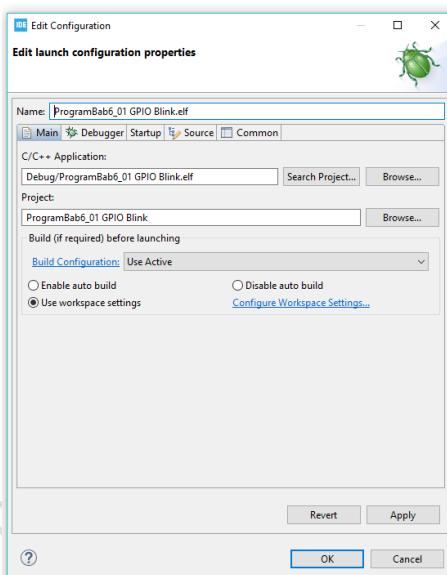
STM32CubeIDE juga menyediakan alat yang dinamakan *Static Stack Analyzer* yang akan menyediakan informasi tentang pemakaian memori stack oleh setiap fungsi dalam program yang dibuat. Pengaturan ukuran memori stack bisa dilakukan pada saat pembuatan proyek melalui STM32CubeMx (*Project Setting*) atau dengan mengedit file linker script. Analisa stack ini akan menampilkan pemakaian memori stack setiap fungsi (*Max Cost*) dan apakah fungsi tersebut akan memanggil fungsi lain (*Depth*) yang tentunya akan mempengaruhi pemakaian stack.



Gambar 6.27 Analisa Stack

Analisa stack ini penting ketika mencari *bug* di program, terutama program yang komplek atau program yang menggunakan RTOS. Dalam RTOS, setiap task (fungsi) telah ditentukan besarnya memori stack. Dengan mengetahui pemakaian stack setiap fungsi bisa diketahui apakah fungsi atau task tersebut akan mengalami overflow stack atau tidak.

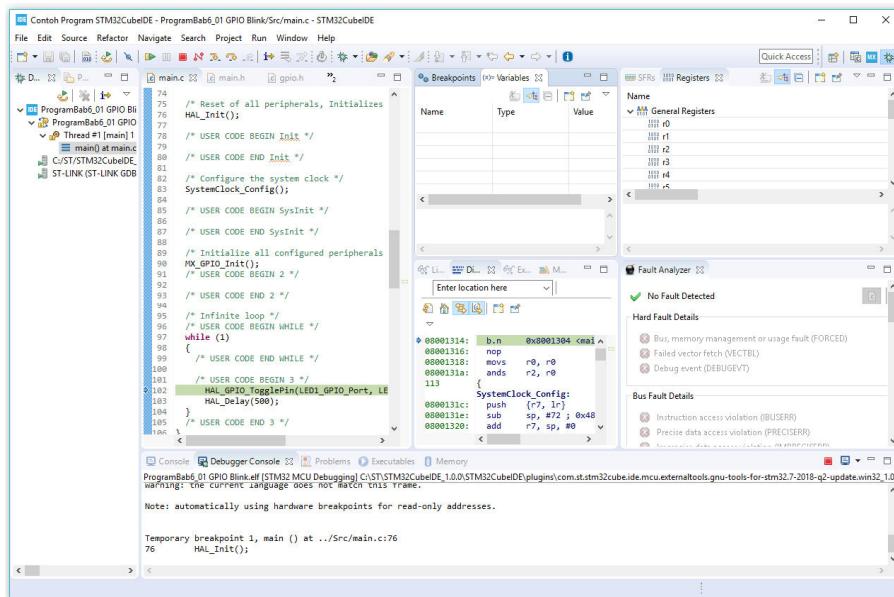
Setelah selesai



Gambar 6.28 Pengaturan Debug

Saat tombol ditekan, STM32CubeIDE akan terhubung ke ST Link dan mengecek apakah mikrokontroler target terhubung ke debugger atau tidak. Kalau terhubung, STM32CubeIDE akan langsung mendownload program ke memori mikrokontroler. Setelah selesai, debugger akan

menjalankan program di mikrokontroler dan berhenti di baris pertama fungsi main.



Gambar 6.29 Proses Debug

Dalam mode debug, program bisa dijalankan per baris. Menjalankan sebuah fungsi yang dipanggil tanpa masuk ke fungsi tersebut atau bisa juga masuk dan menjalankan program di fungsi tersebut baris demi baris. Debug juga memungkinkan untuk melihat isi register ARM Cortex-M3 atau isi SFR (*Special Function Register*) dari periperal yang digunakan atau nilai sebuah variabel yang digunakan dalam program. Mode debug juga akan menampilkan program dalam bahasa assembly (*dissassembly*). Untuk menampilkannya dilakukan melalui menu Window>Show View, lalu pilih SFR, Register atau Dissassembly. STM32CubeIDE juga menyediakan *Fault Analyzer* yang akan mendeteksi terjadinya fault (HardFault, Bus Fault dan seterusnya) dan akan menampilkan baris program yang menyebabkan fault tersebut.

Listing Program 6.6 Program Startup

```
/* Zero fill the bss segment. */
FillZeroBSS:
    movs r3, #0
    str r3, [r2], #4

LoopFillZeroBSS:
```

```

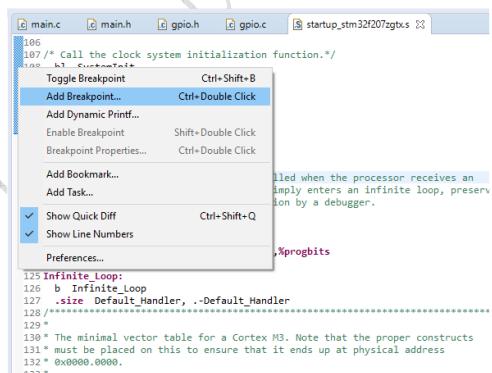
ldr r3, = _ebss
cmp r2, r3
bcc FillZeroBSS

/* Call the clock system initialization function.*/
bl SystemInit
/* Call static constructors */
bl __libc_init_array
/* Call the application's entry point.*/
bl main
bx lr
.size Reset_Handler, .-Reset_Handler

```

Secara default, ketika debug, program akan langsung dimulai dan berhenti di fungsi main. Sebenarnya sebelum memanggil fungsi main, program startup akan memanggil fungsi SystemInit (file `system_stm32f2xx.c`). Agar debugger berhenti di fungsi SystemInit bisa dilakukan dengan penambahan *breakpoint*. Maka setiap kali debug, program akan berhenti di fungsi tersebut. Pada dasarnya script ini akan membuat sebuah *breakpoint*. Fungsi breakpoint ini akan membuat program berhenti di baris yang diberi tanda breakpoint ketika menu *Run/Resume* ditekan.

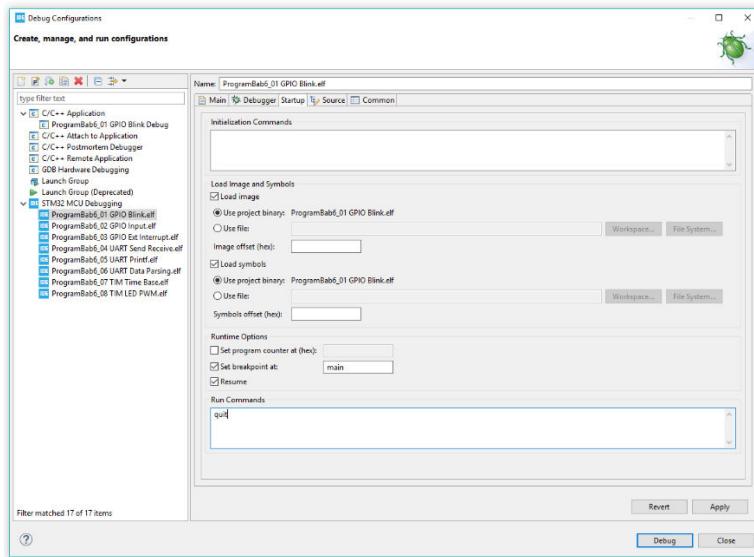
Penambahan breakpoint ini bisa dilakukan di baris program mana saja. Pada baris program yang akan ditandai dengan breakpoint, klik kanan dan klik *Add Breakpoint* dan klik tombol OK ketika *Breakpoint Properties* muncul. Baris program yang telah diberi breakpoint akan ditandai dengan lingkaran kecil berwarna biru.



Gambar 6.30 Penambahan Breakpoint

Breakpoint sangat berguna ketika men-debug program dengan RTOS. Pada RTOS, ketika men-debug dengan baris demi baris ada kemungkinan debugger akan pindah ke task lain, karena pada saat itu *scheduler* melakukan *context switch*, dengan breakpoint debugger bisa diatur agar selalu berhenti di baris program yang diinginkan.

Proses debug ini sangat dibutuhkan saat pengembangan program, tapi ketika program dirasa sudah berjalan sesuai yang diinginkan dan ingin langsung dijalankan tanpa proses debug terlebih dahulu, maka debugger perlu diatur agar setelah men-download program ke memori flash, debugger agar langsung keluar dari mode debug dan STM32F207 akan langsung menjalankan programnya.

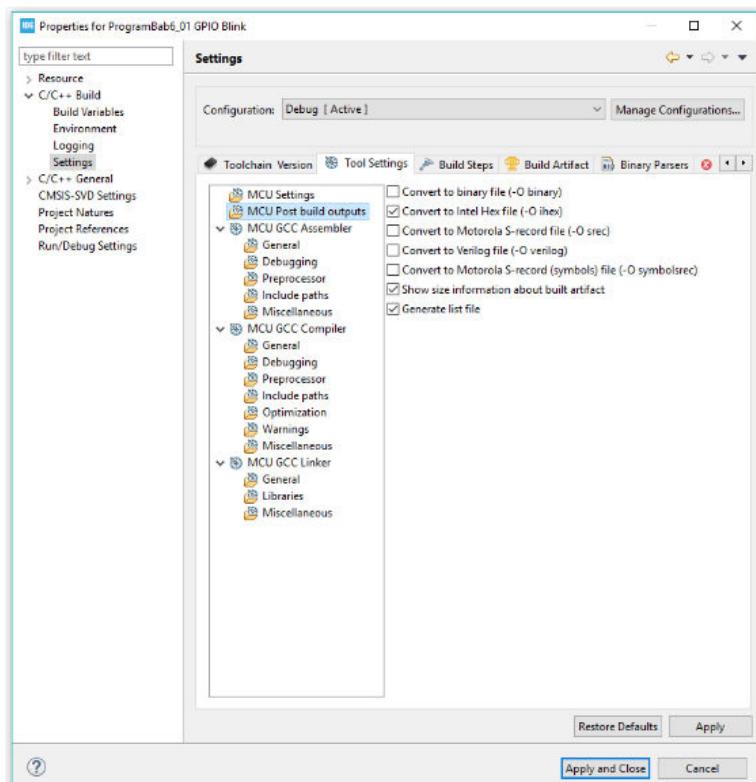


Gambar 6.31 Menjalankan Program Setelah Download

Dari konfigurasi debug, di tab *Startup* bisa ditambahkan perintah *quit* di kotak *Run Commands*. Dengan adanya perintah *quit* ini, setelah debugger mendownload program ke flash memori, debugger akan langsung keluar dari mode debug sehingga STM32F207 akan langsung menjalankan program yang baru diterimanya.

Ketika proses pengembangan program telah selesai dan mungkin akan berlanjut ke proses produksi masal, maka diperlukan output dari program yang dibuat. Karena tidak memungkinkan memprogram di tahap produksi dengan menggunakan IDE. Dari menu properti proyek,

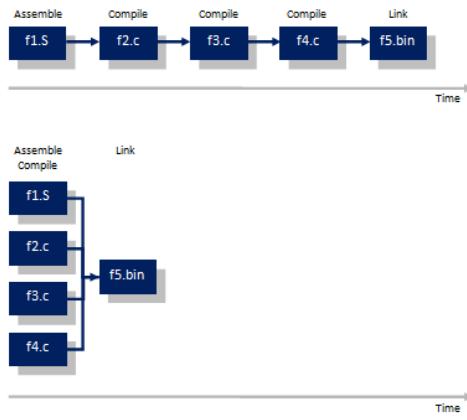
pilih C/C++Build/Setting. Di tab Tool Setting pilih *MCU Post build outputs* dan pilih jenis file output yang akan digunakan, misal Intel Hex atau Binary atau format lainnya. Compile ulang program, maka TrueSTUDIO akan membuat file tersebut. File yang dihasilkan akan disimpan di direktori proyek dalam folder Debug dengan nama sesuai nama proyek dan ekstensi jenis file output yang dipilih, misal .hex untuk file Intel Hex atau .bin untuk file binary.



Gambar 6.32 Pembuatan File Hexa

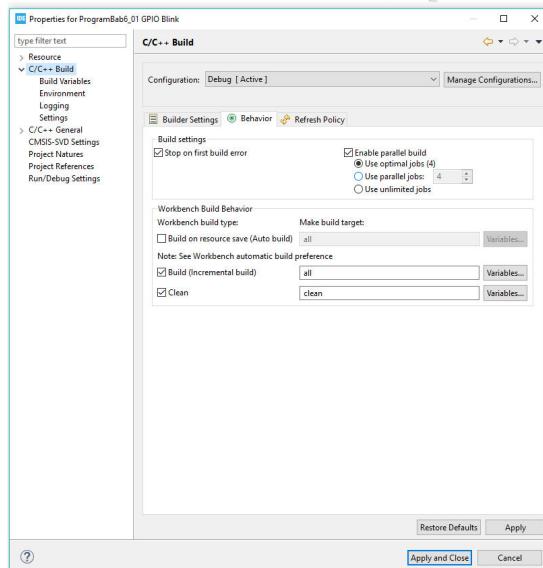
Saat kompilasi, STM32CubeIDE akan meng-compile file source code satu per satu secara berurutan. Ketika proyek semakin besar dengan melibatkan banyak file source code akan mengakibatkan lamanya waktu kompilasi. STM32CubeIDE mempunyai fitur untuk melakukan kompilasi secara paralel. Dengan fitur ini STM32CubeIDE akan meng-compile beberapa file secara bersamaan, sehingga bisa mengurangi waktu kompilasi pada proyek dengan baris dan file source code yang besar.

Kecepatan kompilasi paralel ini tentu saja dipengaruhi oleh kecepatan prosesor komputer yang digunakan. Akses ke hard disk dari aplikasi lain juga bisa memperlambat proses kompilasi paralel ini.



Gambar 6.33 Compile Secara Berutan dan Paralel

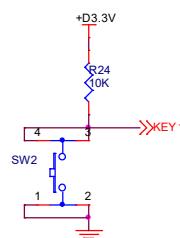
Untuk mengaktifkan kompilasi paralel ini cukup dilakukan dengan mengecek *Enable parallel build* di konfigurasi proyek. Cek juga *Build on resource save (Auto build)* agar setiap kali kompilasi, IDE akan menyimpan terlebih dahulu setiap perubahan dari file source code.



Gambar 6.34 Mode Kompilasi Paralel

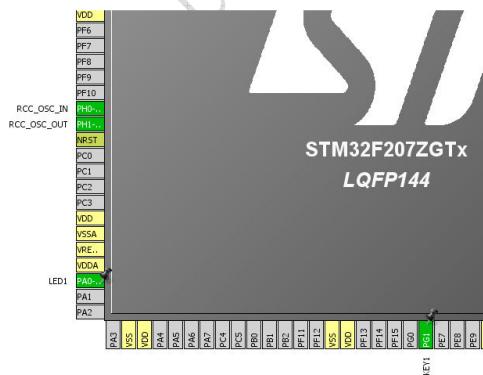
6.1.2 GPIO SEBAGAI INPUT

Salah satu contoh penggunaan GPIO sebagai input adalah membaca saklar. Saklar atau tombol atau *push button* banyak digunakan di perangkat elektronika misal sebagai pengatur volume suara di radio digital. Banyak cara untuk membaca sebuah tombol dan yang paling umum dilakukan adalah dengan membaca kondisi *high* dan *low*, *high* saat saklar tersebut tidak ditekan dan *low* ketika saklar tersebut ditekan. Pada cara ini, saklar biasanya dihubungkan dengan resistor pull-up dan dibaca oleh salah satu pin mikrokontroler yang difungsikan sebagai input.



Gambar 6.35 Rangkain Saklar

Pada contoh program ini, saklar (KEY1) dibaca oleh pin PG1 (GPIOG pin 1). Saklar digunakan untuk mematikan atau menghidupkan sebuah LED yang terhubung ke PA0 setiap kali ditekan. Pembuatan proyek STM32CubeIDE dilakukan seperti pembuatan program sebelumnya, dengan pinout seperti gambar di bawah (LED1 adalah GPIO Output dan KEY1 sebagai GPIO Input).



Gambar 6.36 Pengaturan Pinout di STM32CubeMx

Seperti program sebelumnya, pin LED1 (PA0) difungsikan sebagai output push pull dengan tidak menggunakan pull up atau pun pull down. Nilai LED1 setelah reset adalah logika low (LED mati). Sedangkan pin KEY1 difungsikan sebagai input juga tanpa pull up dan pull down, karena sudah menggunakan pull up eksternal.

Listing Program 6.7 Inisialisasi GPIO

```
void MX_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct;

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOH_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOG_CLK_ENABLE();

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_RESET);

    /*Configure GPIO pin : PtPin */
    GPIO_InitStruct.Pin = LED1_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(LED1_GPIO_Port, &GPIO_InitStruct);

    /*Configure GPIO pin : PtPin */
    GPIO_InitStruct.Pin = KEY1_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(KEY1_GPIO_Port, &GPIO_InitStruct);

}
```

Salah satu hal yang sangat penting untuk diperhatikan ketika membaca saklar adalah efek *bouncing* yang memang selalu ada pada komponen elektro mekanis. Efek bouncing terjadi karena saat sakal itu ditekan, mekanik saklar tidak langsung berada di kondisi on, tetapi akan memantul terlebih dahulu sehingga akan menimbulkan kondisi on/off berulang kali. Dan karena cepatnya mikrokontroler, hal ini akan terbaca sebagai penekanan saklar yang berulang kali yang tentu saja tidak akan sesuai dengan program yang diinginkan. Untuk itu diperlukan penanganan terhadap efek bouncing ini. Cara termudah penanganannya dilakukan di program (software) karena penanganan secara hardware

relatif lebih sulit dan akan menambah *cost* produksi (penambahan komponen).

Listing Program 6.8 Pembacaan Saklar dengan anti bouncing

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
if(!HAL_GPIO_ReadPin(KEY1_GPIO_Port, KEY1_Pin)){
    keypresstimer=0;
    while(!HAL_GPIO_ReadPin(KEY1_GPIO_Port, KEY1_Pin)){
        keypresstimer++;
        HAL_Delay(1);
    }
    if(keypresstimer>10){
        HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
    }
}
/* USER CODE END 3 */
```

Pembacaan saklar dilakukan dengan fungsi HAL_GPIO_ReadPin yang akan bernilai 0 jika saklar ditekan dan bernilai 1 jika saklar tidak ditekan. Ketika saklar ditekan, program akan menunggu sampai saklar dilepas kembali. Selama menunggu, sebuah variabel (*keypresstimer*) digunakan untuk menghitung lamanya penekanan saklar dalam satuan mili detik (dengan memanggil fungsi HAL_Delay). Ketika saklar dilepas, program akan membaca keypresstimer, jika lebih dari 10 (mili detik) program akan men-toggle LED1, jika kurang atau sama dengan 10 milidetik akan diabaikan karena dianggap sebagai bouncing.

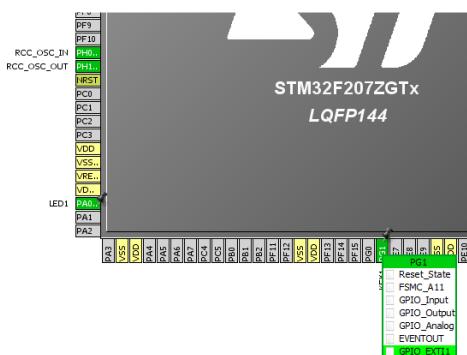
6.1.3 GPIO SEBAGAI SUMBER INTERUPSI

Semua pin GPIO STM32F207 bisa difungsikan sebagai sumber interupsi eksternal. Interupsi bisa diatur saat ada transisi naik (*rising*), transisi turun (*falling*) atau keduanya. Interupsi eksternal melalui pin GPIO dalam STM32F207 diberi nama interupsi EXTI. Oleh karena sebuah GPIO bisa mempunyai sampai 16 pin, akan ada 16 sumber interupsi melalui pin GPIO. Namun vektor interupsi untuk pin GPIO diatur sebagai berikut:

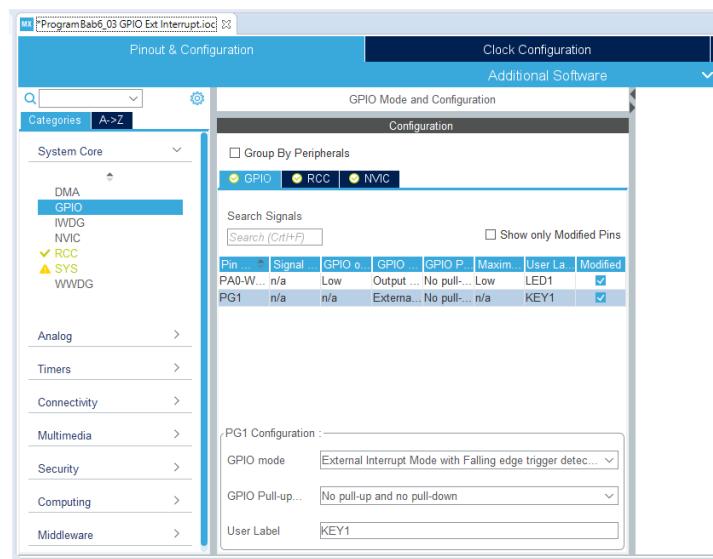
- EXTI0, untuk pin 0

- EXTI1, untuk pin 1
- EXTI2, untuk pin 2
- EXTI3, untuk pin 3
- EXTI4, untuk pin 4
- EXTI9_5, untuk pin 5 – pin 9
- EXTI15_10, untuk pin 10 – pin 15

Jadi hanya pin 0 – pin 4 yang mempunyai vektor interupsi tersendiri. Sedangkan pin 5 – pin 15 mempunyai vektor interupsi bersama. Oleh karena vektor interupsi berdasarkan nomor pin (tidak berdasarkan port), maka tidak bisa mengaktifkan interupsi di pin yang sama secara sekaligus, misal jika PG1 sudah diaktifkan sebagai pin interupsi, maka PA1, PB1 dan seterusnya tidak bisa diaktifkan sebagai interupsi.



Untuk mengaktifkan sebuah pin GPIO sebagai sumber interupsi eksternal, dari tab Pinout STM32CubeMX, pilih pin yang bersangkutan, lalu klik kanan dan pilih GPIO_EXTIx. Di tab Pinout & Configuration di menu System Core/GPIO dan klik di PG1/KEY1, pilih mode GPIO sebagai mode eksternal interupsi, falling atau rising, dalam contoh program kali ini akan dipilih mode falling. Untuk pengaturan prioritas interupsinya menggunakan nilai default yang ditentukan oleh STM32CubeMX, yaitu prioritas 0.



Gambar 6.38 Mode Interupsi Eksternal

Listing Program 6.9 Inisialisasi GPIO sebagai interupsi

```

/*Configure GPIO pin : PtPin */
GPIO_InitStruct.Pin = KEY1_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(KEY1_GPIO_Port, &GPIO_InitStruct);

/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI1 IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI1 IRQn);

```

Sebuah interupsi tentunya membutuhkan sebuah fungsi untuk melayani ketika interupsi itu terjadi. Program akan memanggil atau melompat ke fungsi tersebut, kemudian setelah selesai akan kembali ke alamat program di mana interupsi terjadi. STM32CubeMX akan menempatkan fungsi-fungsi layanan interupsi dalam sebuah file stm32f2xx_it.c. Setiap kali pembuatan proyek dengan STM32CubeMX, file ini akan selalu dibuat walaupun program tidak mengaktifkan interupsi periperal. Hal ini karena di ARM Cortex-M3 selain interupsi yang dibangkitkan oleh periperal, ada juga interupsi atau eksepsi yang dibangkitkan oleh core CPU, yaitu eksepsi Hardfault dan sebagainya. Selain itu STM32CubeMX juga akan mengaktifkan interupsi timer SysTick untuk pewaktuan, misal untuk fungsi delay.

Nama-nama fungsi layanan interupsi telah ditentukan dalam file startup (startup_stm32f207xx.s). untuk interupsi eksternal melalui pin 1 dinamakan EXTI1_IRQHandler.

Listing Program 6.10 Fungsi Layanan Interupsi Pin 1

```
void EXTI1_IRQHandler(void)
{
    /* USER CODE BEGIN EXTI1_IRQHandler_0 */

    /* USER CODE END EXTI1_IRQHandler_0 */
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_1);
    /* USER CODE BEGIN EXTI1_IRQHandler_1 */

    /* USER CODE END EXTI1_IRQHandler_1 */
}
```

Fungsi EXTI1_IRQHandler ini akan memanggil fungsi yang akan melayani interupsi GPIO yaitu HAL_GPIO_EXTI_IRQHandler. Sebenarnya bisa juga menambahkan program di dalam fungsi layanan interupsi yang dibuat oleh STM32CubeMX. Namun dalam contoh program ini, layanan interupsi akan mengikuti seperti yang dibuat oleh STM32CubeMX.

Listing Program 6.11 Fungsi HAL_GPIO_EXTI_IRQHandler

```
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
    /* EXTI line interrupt detected */
    if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET)
    {
        __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
        HAL_GPIO_EXTI_Callback(GPIO_Pin);
    }
}
```

Hal pertama yang dilakukan dalam layanan interupsi ini adalah memanggil sebuah makro (`_HAL_GPIO_EXTI_GPIO_IT`). Makro ini akan mencek register EXTI_PR (*Pending Register*) yang berisi bit-bit status interupsi yang akan di-set ketika interupsi yang bersangkutan terjadi, dalam hal ini bit ke-1. Bit tersebut kemudian harus di-reset untuk mengijinkan interupsi selanjutnya tentunya setelah interupsi yang sekarang selesai dilayani. Hal ini dilakukan dengan memanggil sebuah makro `_HAL_GPIO_EXTI_CLEAR_IT`. Layanan interupsi selanjutnya memanggil fungsi HAL_GPIO_EXTI_Callback.

Fungsi ini sebenarnya hanya pilihan. Oleh karena itu, STM32CubeMX mendefinisikan fungsi ini dengan keyword `_weak`. Dalam bahasa C,

keyword `_weak` artinya saat compilasi, compiler akan mencari fungsi tersebut di semua file sumber atau pustaka yang dipakai dalam proyek. Jika tidak ditemukan, maka compiler akan mencari fungsi dengan keyword `_weak` tersebut. Jika tidak ditemukan juga baru compiler akan membangkitkan error.

Listing Program 6.12 Fungsi Callback `_weak`

```
_weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    /* Prevent unused argument(s) compilation warning */
    UNUSED(GPIO_Pin);
    /* NOTE: This function Should not be modified, when the callback
is needed,
       the HAL_GPIO_EXTI_Callback could be implemented in the
user file
    */
}
```

Fungsi callback sebenarnya dideklarasikan di file main.c

Listing Program 6.13 Fungsi callback sebenarnya

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){
    keyint=1;
}
```

Dalam fungsi callback ini program hanya menunjukkan bahwa telah terjadi interupsi di pin PG1 dengan menset variabel keyint yang dideklarasikan sebagai variabel global.

Listing Program 6.14 Fungsi main

```
while (1)
{
    /* USER CODE END WHILE */

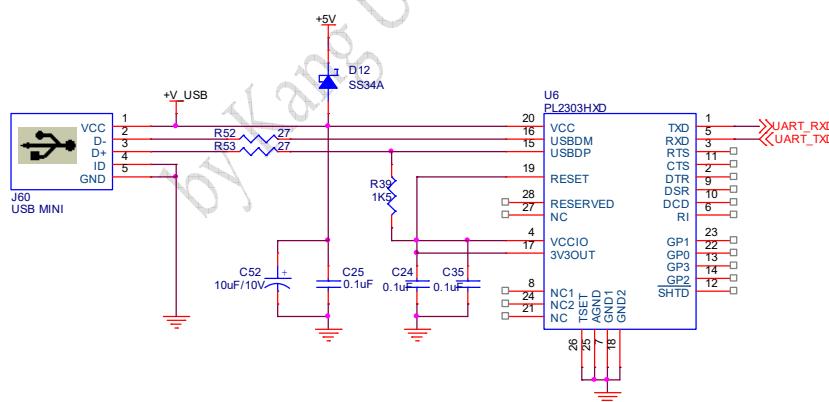
    /* USER CODE BEGIN 3 */
    if(keyint){
        keypresstimer=0;
        while(!HAL_GPIO_ReadPin(KEY1_GPIO_Port, KEY1_Pin)){
            keypresstimer++;
            HAL_Delay(1);
        }
        if(keypresstimer>10){
            HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
        }
    }
}
```

Di fungsi loop utama, program hanya mengecek variabel keyint tersebut, jika diset maka program akan men-toggle LED dan mereset variabel keyint dan memanggil fungsi delay sebagai anti bouching. Oleh karena interupsi diset di mode *falling* maka interupsi terjadi ketika tombol ditekan (LED toggle saat tombol ditekan). Ketika mode interupsi diganti menjadi *rising*, maka interupsi terjadi ketika tombol dilepas.

6.2 BERMAIN-MAIN DENGAN UART

UART (port serial) banyak digunakan sebagai sarana antarmuka komunikasi dengan komputer, modem GSM/3G, modul bluetooth, modul wifi, sensor-sensor industri dan sebagainya. Yang membedakan adalah protokol komunikasinya, misal modem, bluetooth, dan wifi menggunakan *AT Command*, sedangkan sensor-sensor industri menggunakan protokol Modbus. STM32F207 sendiri dilengkapi dengan 6 buah UART, 4 di antaranya mendukung USART (sinkron).

Dalam contoh program yang akan dibahas di sub-bab ini, akan dijelaskan bagaimana menggunakan USART1 untuk berkomunikasi dengan PC/Laptop dan teknik parsing data untuk menyalakan beberapa buah LED melalui komputer. PC/Laptop generasi terbaru sudah tidak mempunyai port serial (RS-232), dan sudah diganti dengan port USB. Oleh karena itu agar port serial STM32F207 bisa terhubung dengan PC, diperlukan sebuah IC USB ke serial, misalnya PL2303HX dari Prolific (seperti gambar di bawah). Tentu saja driver untuk IC ini harus terinstal ke PC/Laptop.

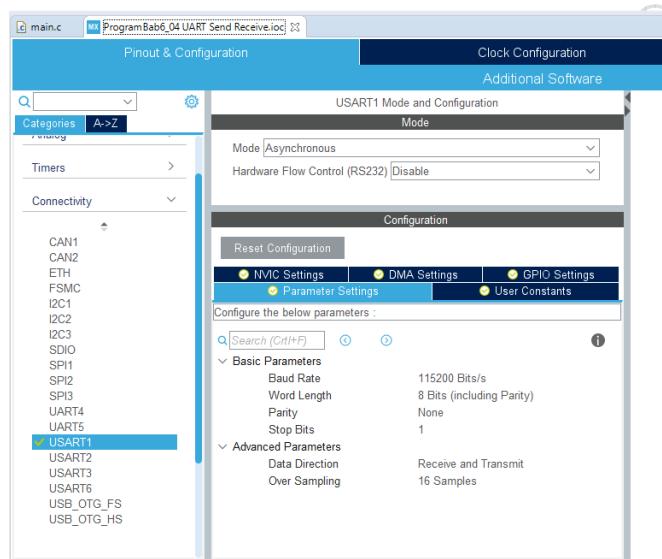


Gambar 6.39 Skematik USB ke Serial PL2303HX

Hubungan pin RXD dan TXD antara PL2303HXD dengan STM32F207 harus silang (*cross*), artinya pin TXD PL2303 (pin 1) terhubung dengan pin PA10 (USART1_RXD) STM32F207 (pin ke 102 di kemasan LQFP144) dan pin RXD PL2303 (pin 5) harus terhubung dengan pin PA9 (USART1_TXD) STM32F207 (pin ke 101 di kemasan LQFP144). Begitu juga ketika port serial dihubungkan dengan modem, bluetooth dan lain-lain, koneksi harus silang.

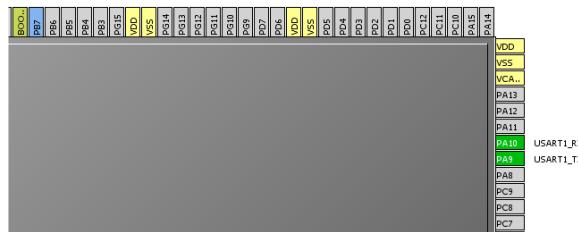
6.2.1 KIRIM DAN TERIMA DATA SERIAL

USART1 diaktifkan melalui STM32CubeMX dengan memilih mode UART *Asynchronous*. Pin-pin yang mempunyai fungsi alternatif USART1 akan berubah warnanya menjadi hijau, secara default STM32CubeMX akan memilih PA9 (RXD) dan PA10 (TXD) sebagai pin untuk USART1, kecuali pin tersebut telah dipakai.



Gambar 6.40 Pemilihan USART1

Untuk melihat pin alternatif lain, bisa dilakukan dengan cara mengklik pin tersebut sambil menekan tombol [CTRL] di keyboard. Jika ada pin alternatif, maka pin tersebut akan berubah menjadi biru ketika pengklikan dilakukan. Misal untuk pin USART1_TXD pin alternatifnya selain PA10 adalah PB7, dengan catatan PB7 belum aktifkan sebagai fungsi periperal yang lain.



Gambar 6.41 Pin Alternatif USART1

Parameter yang paling utama yang harus disesuaikan antara kedua perangakat (PC dan STM32) adalah baud rate, jumlah bit data, bit paritas, dan bit stop. Pada program kali ini pengaturan diatur baud rate 115200 bps, 8 bit data, 1 bit stop dan tanpa bit paritas. Parameter ini harus sama antara PC dan STM32. Kalau tidak, komunikasi serial tidak akan terjadi karena komunikasi serial UART tidak menggunakan sinyal clock sinkronisasi, pengirim dan penerima akan menterjemahkan data dari pewaktuan dari aliran bit data yang diterima, yang tergantung pada parameter-parameter tersebut. Di contoh program ini, tidak digunakan interupsi atau pemakaian DMA.

Untuk mengaktifkan sebuah periperal, maka perlu dilakukan 2 pengaturan, yaitu periperal itu sendiri dan GPIO yang dipetakan sebagai fungsi alternatif untuk periperal tersebut. Jadi walapun program tidak menggunakan GPIO sebagai pin input/output biasa, STM32CubeMX akan selalu menghasilkan fungsi `MX_GPIO_Init()`, walaupun isinya hanya mengaktifkan clock untuk GPIO yang dipakai oleh periperal yang diaktifkan.

Listing Program 6.15 Fungsi Init GPIO

```
void MX_GPIO_Init(void)
{
    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOH_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();

}
```

USART1 sendiri di-inisialisasi melalui fungsi `MX_USART1_UART_Init()`.

Listing Program 6.16 Fungsi Init USART1

```
void MX_USART1_UART_Init(void)
{
```

```

huart1.Instance = USART1;
huart1.Init.BaudRate = 115200;
huart1.Init.WordLength = UART_WORDLENGTH_8B;
huart1.Init.StopBits = UART_STOPBITS_1;
huart1.Init.Parity = UART_PARITY_NONE;
huart1.Init.Mode = UART_MODE_TX_RX;
huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
huart1.Init.OverSampling = UART_OVERSAMPLING_16;
if (HAL_UART_Init(&huart1) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}
}

```

Sedangkan inisialisasi GPIO sendiri dilakukan ketika fungsi inisialisasi USART1 memanggil fungsi HAL_UART_Init().

Listing Program 6.17 Fungsi Inisialisasi GPIO untuk USART1

```

void HAL_UART_MspInit(UART_HandleTypeDef* uartHandle)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    if(uartHandle->Instance==USART1)
    {
        /* USER CODE BEGIN USART1_MspInit 0 */

        /* USER CODE END USART1_MspInit 0 */
        /* USART1 clock enable */
        __HAL_RCC_USART1_CLK_ENABLE();

        /**USART1 GPIO Configuration
        PA9      -----> USART1_TX
        PA10     -----> USART1_RX
        */
        GPIO_InitStruct.Pin = GPIO_PIN_9|GPIO_PIN_10;
        GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
        GPIO_InitStruct.Pull = GPIO_PULLUP;
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
        GPIO_InitStruct.Alternate = GPIO_AF7_USART1;
        HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

        /* USER CODE BEGIN USART1_MspInit 1 */

        /* USER CODE END USART1_MspInit 1 */
    }
}

```

Pada contoh program UART yang pertama ini, hanya akan menunggu sebuah karakter yang dikirim oleh sebuah program terminal serial yang

jalan di PC dan kemudian mengirimkannya kembali. Setelah proses inisialisasi, program diawali dengan perintah untuk menekan sebuah tombol di keyboard PC melalui program terminal. Dengan mengirimkan kalimat “Serial Port Test\r\n” kemudian “Type any key in terminal\r\n”.

Listing Program 6.18 Program Kirim dan Terima Data Serial

```
/* USER CODE BEGIN 2 */
HAL_UART_Transmit(&huart1,(uint8_t*)"Serial Port Test\r\n", 18,
1000);
HAL_UART_Transmit(&huart1,(uint8_t*)"Type any key in
terminal\r\n", 26, 1000);
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    if (HAL_UART_Receive(&huart1,aRxBuffer,1,1000)==HAL_OK){
        HAL_UART_Transmit(&huart1,(uint8_t*)"Receive:",8,500);
        HAL_UART_Transmit(&huart1,aRxBuffer,1,1000);
        HAL_UART_Transmit(&huart1,(uint8_t*)"\\r\\n",2,1000);
    }
    /* USER CODE END 3 */
}
```

Library HAL telah menyediakan fungsi-fungsi untuk mengirim dan menerima data port serial, tanpa atau dengan interupsi, tanpa atau dengan menggunakan DMA. Pada contoh program kali ini akan dipergunakan fungsi-fungsi tanpa interupsi dan tanpa DMA.

Untuk mengirim data, HAL menyediakan fungsi:

Listing Program 6.19 Fungsi Mengirim Data Serial

```
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart,
uint8_t *pData, uint16_t Size, uint32_t Timeout)
```

Fungsi ini membutuhkan beberapa parameter:

- UART_HandleTypeDef *huart, yang merupakan sebuah pointer ke periperal UART yang dimaksud (huart1).
- uint8_t *pData, menunjuk ke pointer dari buffer data yang akan dikirim.

- `uint16_t size`, ukuran buffer yang akan dikirim (byte atau karakter).
- `uint32_t timeout`, waktu tunggu karena ketika mengirim data ke port serial harus melihat status dari port serial tersebut. Jika status tidak sesuai dan telah melebihi waktu timeout, maka program bisa kembali dan tidak *hang* di fungsi tersebut.

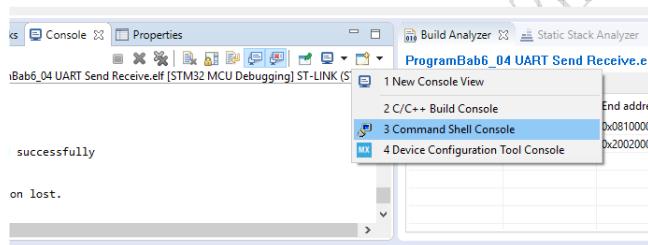
Sedangkan untuk menerima data HAL, menyediakan fungsi:

Listing Program 6.20 Fungsi Menerima Data UART

```
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart,
uint8_t *pData, uint16_t Size, uint32_t Timeout)
```

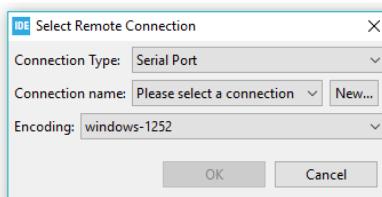
Parameter untuk mengirim dan menerima hampir sama, bedanya di buffer. Pada saat menerima, data akan disimpan di pData.

Dalam contoh program kali ini, serial port hanya diprogram untuk menerima data 1 karakter, begitu menerima, data langsung dikirim kembali ke port serial tersebut. Program terminal, bisa menggunakan terminal yang telah disediakan oleh STM32CubeIDE, dari jendela Console, pilih *Comand Shell Console*. Kemudian klik icon bergambar monitor untuk mengatur terminal (port serial).



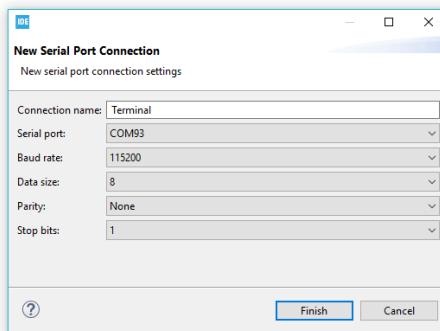
Gambar 6.42 Menampilkan Terminal

Dari jendela *Select Remote Connection* Pilih *Connection Type* Serial Port. Jika sebelumnya sudah pernah membuat koneksi serial, dari *Connection name* pilih koneksi yang pernah dibuat, jika belum maka harus membuat koneksi baru dengan mengklik tombol New.

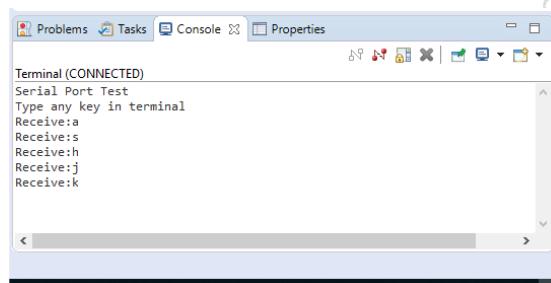


Gambar 6.43 Pembuatan Koneksi Baru

Setelah terminal diatur disesuaikan dengan pengaturan UART1 di program STM32, terminal akan segera menampilkan data yang dikirim dari UART1. Dan setiap kali menekan tombol keyboard di terminal, data karakter yang dikirim oleh terminal, akan dikirim balik oleh program STM32.



Gambar 6.44 Pengaturan Port Serial



Gambar 6.45 Program Kirim dan Terima Serial

Ada masalah dengan fungsi yang meminta panjang data yang telah ditentukan, terutama saat penerimaan data. Untuk pengiriman, program pastinya tahu berapa panjang data yang harus dikirim. Sedangkan untuk penerimaan, program sering kali tidak tahu berapa panjang data yang harus diterimanya. Jika data yang diterima kurang dari panjang data yang telah ditentukan, maka program akan menunggu sampai waktu timeout habis, sedangkan jika data yang diterima lebih banyak dari panjang data yang ditentukan, maka akan ada data yang hilang. Teknik tersendiri diperlukan untuk menangani hal ini, seperti pada contoh program berikutnya.

6.2.2 FUNGSI PRNTF KE PORT SERIAL

Dalam banyak aplikasi, pengiriman data melalui port serial menggunakan format ASCII sehingga data bisa langsung terbaca melalui program terminal. Namun ketika program memerlukan pengiriman data sebuah variabel dalam bentuk ASCII, fungsi pengiriman data yang disediakan oleh HAL tidak bisa langsung melakukannya. Diperlukan fungsi tambahan untuk mengkonversi data tersebut ke dalam format ASCII.

Bahasa C, telah menyediakan sebuah fungsi untuk melakukan hal itu, yaitu fungsi *printf*. Dalam pemrograman PC (desktop) fungsi printf akan langsung diarahkan ke monitor. Dalam sistem embedded, fungsi printf bisa diarahkan ke UART, I2C, LCD, atau pun ke sebuah port GPIO. Program perlu menyediakan sebuah fungsi yang nantinya akan dipanggil oleh fungsi printf untuk mengirimkan byte demi byte data ke port atau IO yang dimaksud.

Fungsi printf membutuhkan sebuah fungsi (*_write*) yang digunakan untuk mengarahkan ke mana data akan dikirim. Fungsi *_write* ini berada di file *syscall.c*, yang dihasilkan setiap kali membuat proyek dengan STM32CubeIDE.

Listing Program 6.21 Fungsi write

```
attribute((weak)) int _write(int file, char *ptr, int len){
    int DataIdx;
    for (DataIdx = 0; DataIdx < len; DataIdx++)
    {
        __io_putchar( *ptr++ );
    }
    return len;
}
```

Selanjutnya perlu didefinisikan fungsi *__io_putchar* yang akan mengarahkan ke mana data akan dikirim. Oleh karena data akan dikirim melalui UART, ada baiknya fungsi putchar tersebut diletakan di file *uart.c*. sisipkan terlebih dahulu deklarasi fungsi putchar di file *uart.c* Compiler GCC (GNU) menggunakan nama fungsi *__io_putchar* (seperti telah ditentukan di fungsi *_write*) beberapa menggunakan nama *fputc*.

Listing Program 6.22 Deklarasi PUTCHAR

```
/* USER CODE BEGIN 0 */
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
```

```
/* USER CODE END 0 */
```

Fungsi putchar sendiri mengarahkan pengiriman ke USART1 dan digunakan fungsi HAL_UART_Transmit dengan panjang data 1.

Listing Program 6.23 Fungsi PUTCHAR ke UART

```
/* USER CODE BEGIN 1 */
PUTCHAR_PROTOTYPE
{
    /* Place your implementation of fputc here */
    /* e.g. write a character to the USART */
    HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xFFFF);
    return ch;
}
/* USER CODE END 1 */
```

Selain itu fungsi write mempunyai atribut weak, sehingga bisa ditimpak. Fungsi write pada dasarnya akan mengirimkan sejumlah data dalam bentuk char sebanyak len melalui fungsi __io_putchar melalui perulangan. Fungsi __io_putchar sendiri akan memanggil fungsi HAL_UART_Transmit. Padahal fungsi HAL_UART_Transmit bisa mengirim data lebih dari 1, sehingga sebenarnya fungsi write tidak perlu menggunakan perulangan __io_putchar, tapi langsung memanggil fungsi HAL_UART_Transmit dan mengirimkan data dari pointer ptr secara langsung.

Listing Program 6.24

```
int _write(int file, char *ptr, int len){
    HAL_UART_Transmit(&huart1, (uint8_t *)ptr, len, 0xFFFF);

    return len;
}
```

Setelah deklarasi fungsi putchar atau fungsi write, maka fungsi printf siap untuk digunakan. Pastikan file stdio.h disertakan ke dalam program (#include). Deklarasikan variabel-variabel berikut di fungsi main.

Listing Program 6.25 Deklarasi Variabel

```
/* USER CODE BEGIN 1 */
uint16_t a;
int16_t b;
float c;
uint8_t d;
uint8_t buffer[]="Serial printf example";
uint32_t e;
/* USER CODE END 1 */
```

Dan tambahkan program berikut:

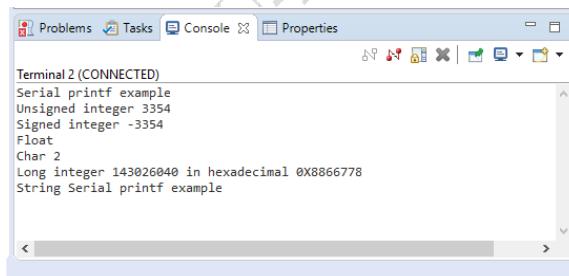
Listing Program 6.26 Contoh Program printf

```
/* USER CODE BEGIN 2 */
a = 3354;
b = -3354;
c = 3.554;
d = '2';
e = 0x8866778;

printf("Serial printf example\r\n");
printf("Unsigned integer %d\r\n",a);
printf("Signed integer %d\r\n",b);
printf("Float %.3f\r\n", c);
printf("Char %c\r\n",d);
printf("Long integer %lu in hexadecimal %#04X\r\n",e,e);
printf("String %s\r\n",buffer);
/* USER CODE END 2 */
```

Fungsi printf memerlukan tipe data yang akan diformat ke bentuk string (ASCII), dalam contoh program ini digunakan beberapa deklarasi tipe data:

- %d, untuk tipe data integer 16 bit (baik bertanda maupun tidak).
- %.3f atau %f, untuk tipe data float (pecahan), angka 3 menunjukkan angka dibelakang koma.
- %c, untuk tipe data char atau byte
- %lu, untuk tipe data 32 bit (bertanda maupun tidak)
- %#04X, untuk tipe bilangan heksadesimal
- %s, untuk tipe data string

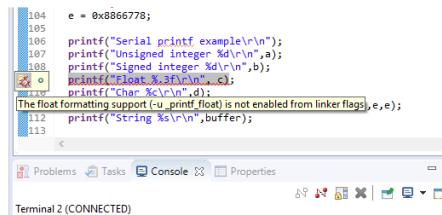


Gambar 6.46 Hasil Program printf

Hasil program ketika dilihat melalui terminal menunjukkan semua tipe data telah terformat dengan benar, kecuali untuk tipe data float. Hal ini karena pada saat compile STM32CubeIDE menggunakan *Runtime Library Reduced C* yang secara default mematikan fungsi format float. Hal ini

karena fungsi printf terutama untuk data float memerlukan memori (RAM) yang cukup banyak.

STM32CubeIDE juga memberikan tanda error pada baris kode printf yang menggunakan format float, walaupun program masih tetap bisa di-compile.



```

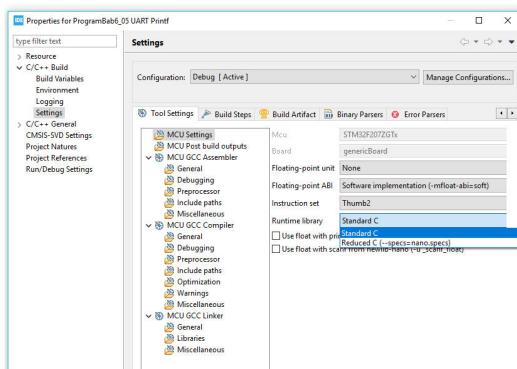
104     e = 0x8866778;
105
106     printf("Serial printf example\r\n");
107     printf("Unsigned integer %d\r\n",a);
108     printf("Signed integer %d\r\n",b);
109     printf("float %.3f\r\n",c);
110     printf("char %c\r\n",d);
111     printf("String %s\r\n",buffer);
112
113

```

The float formatting support (-u_printf_float) is not enabled from linker flags, e,e;

Gambar 6.47 Error Printf Format Float

Untuk mengaktifkan format float bisa dilakukan dengan mengganti Reduced C menjadi Standard C. Klik kanan di proyek Serial Printf dan klik Properties. Di C/C++ Build/Setting pilih tab Tool Setting dan klik MCU Setting. Dan dari Runtime Library pilih Standard C. Lalu compile ulang proyek dan program ke STM32F207.



Gambar 6.48 Pemilihan Library Runtime

Setelah menggunakan Standard C, variabel float bisa dikirim melalui fungsi printf.

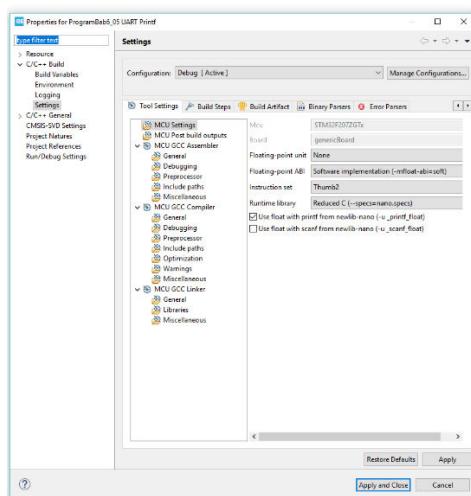
```

Terminal (CONNECTED)
Serial printf example
Unsigned integer 3354
Signed integer -3354
Float 3.559
Char 2
Long integer 143026040 in hexadecimal 0X8866778
String Serial printf example

```

Gambar 6.49 *Printf dengan Format Float*

Penggunaan library Standard C akan menggunakan RAM lebih banyak dibandingkan dengan Reduced C. Untuk mengaktifkan format float di Reduced C cukup mudah, tinggal cek menu *Use float with printf from newlib-nano (-u_printf_float)*. Perbedaan penggunaan memori RAM antara Standard C dan Reduced C cukup signifikan. Pada saat menggunakan Standard C program akan menggunakan RAM sebesar 4,09KB (3,19%) sedangkan saat menggunakan Reduced C penggunaan RAM sebesar 2,07KB (1,62%), berkurang sekitar 50%.



Gambar 6.50 *Cara Kedua Mengaktifkan Format Float*

6.2.3 TEKNIK PARSING DATA

Dalam contoh-contoh program sebelumnya, belum ada interaksi yang berarti antara PC dan STM32F207. Dalam contoh pertama, memang sudah ada interaksi, tetapi STM32 hanya mengirimkan kembali karakter-karakter yang diterimanya (*loop back*). Di contoh program berikut ini, PC akan digunakan untuk meminta STM32F207 untuk melakukan sesuatu,

yaitu mengendalikan 5 buah LED yang terhubung ke STM32F207. PC akan mengendalikan masing-masing LED untuk nyala, mati atau berkedip dengan kecepatan yang bisa diatur.

Kelima LED terhubung ke pin STM32F207 seperti ditunjukkan oleh tabel di bawah. USART1 tetap digunakan untuk berkomunikasi dengan PC dengan baud rate sama dengan program sebelumnya. Pada contoh program kali ini, interupsi penerimaan USART1 diaktifkan. Data yang dikirim dari PC akan diterima melalui interupsi serial, namun karena panjang data yang diterima tidak tetap, maka tidak akan digunakan fungsi penerimaan data yang disediakan oleh HAL. Data yang dikirim dari PC, akan disimpan dulu di buffer data, setelah data lengkap baru diproses.

Tabel 6.1 Koneksi LED

LED	Port	Pin
LED1	GPIOA	Pin 0 (PA0)
LED2	GPIOA	Pin 3 (PA3)
LED3	GPIOB	Pin 0 (PB0)
LED4	GPIOB	Pin 1 (PB1)
LED5	GPIOF	Pin 9 (PF9)

Contoh program yang dinamakan dengan *Data Parsing* ini terdiri atas 3 bagian, yaitu bagian yang mengendalikan kelima 5, bagian yang memproses data yang dikirim dari PC, dan bagian yang menerima data dari PC. Bagian pertama dan kedua, berjalan di program utama, sedangkan bagian ketiga, oleh karena penerimaan data dilakukan melalui interupsi, maka bagian ini berjalan secara *background* dan ketika menerima data frame dari PC, akan memberi sinyal ke program utama sehingga data bisa segera diproses.

Listing Program 6.27 Program Utama

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_USART1_UART_Init();
/* USER CODE BEGIN 2 */
    HAL_UART_ENABLE_IT(&huart1, UART_IT_RXNE);
    LED_Default();
/* USER CODE END 2 */
```

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    LED_Operation();
    if(pcframe){
        pc_link();
    }
}
/* USER CODE END 3 */
```

Ketika mengaktifkan interupsi sebuah periperal, maka ada dua tahap yang harus dilakukan, yaitu mengaktifkan flag interupsi di kendali interupsi (NVIC) sesuai dengan nomor eksepsinya, dan mengaktifkan kendali interupsi di periperal yang bersangkutan, misal untuk USART apakah interupsi diaktifkan saat pengiriman atau penerimaan data. Saat inisialisasi USART, melalui fungsi MX_USART_Init(), hanya melakukan aktivasi interupsi di register NVIC. Library HAL menyediakan fungsi pengiriman dan penerimaan data UART dengan interupsi, HAL_UART_Transmit_IT dan HAL_UART_Receive_IT. Kedua fungsi tersebut, seperti pengiriman dan penerimaan tanpa interupsi, memerlukan panjang data yang telah ditentukan. Sedangkan di contoh program ini, data yang akan diterima panjangnya bisa bervariasi. Dan hanya interupsi penerimaan yang diaktifkan.

Oleh karena itu, interupsi penerimaan data serial perlu dilakukan dengan memanggil sebuah makro yang telah disediakan oleh HAL:

Listing Program 6.28 Mengaktifkan Interupsi Penerimaan UART

```
__HAL_UART_ENABLE_IT(&huart1, UART_IT_RXNE);
```

Begitu juga fungsi layanan interupsi penerimaan harus dilayani dengan fungsi yang tidak disediakan oleh library HAL yaitu fungsi HAL_UART_IRQHandler. Fungsi layanan interupsi hanya menerima data dan menyimpannya ke sebuah buffer data (pcbuffer) dan ketika menerima karakter ENTER (0x0D) program di interupsi akan memberi sinyal ke program utama bahwa telah diterima 1 frame data dengan men-set variabel pcframe. Program layanan interupsi ini diawali dengan membaca register status (SR) bit RXNE yang jika di-set menandakan bahwa register data (DR) telah berisi data. Lalu membaca apakah sumber interupsi adalah penerimaan data atau bukan (register kendali CR1 bit

RXNEIE). Kalau semua kondisi terpenuhi data di register DR dibaca dan disimpan di buffer. Kalau data tersebut bernilai 0x0D, variabel pcframe akan di set.

Listing Program 6.29 Fungsi Layanan Interupsi USART1

```
void USART1_IRQHandler(void)
{
    /* USER CODE BEGIN USART1_IRQHandler_0 */
    uint32_t tmp_flag = 0, tmp_it_source = 0;
    __IO uint16_t dummy;

    /* USER CODE END USART1_IRQHandler_0 */
    //HAL_UART_IRQHandler(&huart1);
    /* USER CODE BEGIN USART1_IRQHandler_1 */
    tmp_flag = __HAL_UART_GET_FLAG(&huart1, UART_FLAG_RXNE);
    tmp_it_source = __HAL_UART_GET_IT_SOURCE(&huart1, UART_IT_RXNE);
    /* UART in mode Receiver -----*/
    if((tmp_flag != RESET) && (tmp_it_source != RESET))
    {
        dummy= huart1.Instance->DR;
        if (pcbuffercounter<sizeof(pcbuffer)&&(dummy!=13)&&(dummy!=10))
            pcbuffer[pcbuffercounter++]=dummy;
        if(dummy==13){
            pcframe=1;
        }
    }
    /* USER CODE END USART1_IRQHandler_1 */
}
```

Di program utama, kelima LED dikendalikan melalui fungsi LED_Operation. Setiap LED mempunyai kendalinya masing-masing dengan menggunakan tipe data struktur LEDTypeDef:

Listing Program 6.30 Kendali LED

```
typedef struct{

    uint8_t status;
    uint32_t timer;
    uint8_t mode;
    uint16_t blinkrate;

}LEDTypeDef;
```

Variabel status akan menyimpan status LED apakah sedang nyala (LED_STATUS_ON) atau sedang mati (LED_STATUS_OFF), timer akan menyimpan timer ketika LED sedang berkedip. Variabel mode akan menentukan apakah LED beroperi di mode ON/OFF

(LEDMODE_ONOFF) atau di mode berkedip (LEDMODE_BLINK) yang kecepatan berkedipnya ditentukan oleh variabel blinkrate (dalam milidetik). Sesaat setelah program berjalan, kelima LED ini ditentukan modenya masing-masing (LED_Default), LED 1,2,3, dan 5 bekerja di mode berkedip dengan kecepatan yang berbeda-beda, sedangkan LED 4 bekerja di mode ON/OFF dengan LED nyala.

Pengendalian LED ini dilakukan dengan membaca mode dari masing-masing LED, ketika mode ON/OFF LED akan diatur sesuai dengan nilai statusnya dengan fungsi HAL_GPIO_WritePin sedangkan jika mode LED berkedip LED diatur dengan fungsi HAL_GPIO_TogglePin melalui fungsi set_LED dengan kecepatan ditentukan oleh blinkrate-nya. Pada saat mode ON/OFF, maka variabel timer dan blinkrate akan diabaikan oleh program dan ketika mode berkedip, variabel status yang diabaikan.

Listing Program 6.31 Fungsi Pengendali LED

```
void LED_Operation(void){  
    uint8_t i;  
  
    for(i=0;i<5;i++){  
        if(LED[i].mode==LEDMode_BLINK){  
            if(HAL_GetTick() - LED[i].timer>LED[i].blinkrate){  
                set_LED(i,LEDMode_BLINK,0);  
                LED[i].timer = HAL_GetTick();  
            }  
        }  
        else{  
            set_LED(i,LEDMode_ONOFF,LED[i].status);  
        }  
    }  
}
```

Untuk mengganti mode atau mengatur kecepatan berkedip LED atau menyalakan dan mematikan LED dilakukan dengan mengirim perintah melalui port serial (USART1). Seperti telah dijelaskan, perintah ini diterima melalui interupsi USART1 dan disimpan di pcbuffer. Ketika data telah diterima lengkap 1 frame, akan memberi sinyal ke program utama untuk memproses perintah tersebut (fungsi pclink).

Format data atau perintah yang dikirimkan dari PC harus didefinisikan. Format data ini adalah protokol komunikasi yang mengatur antara PC dan MCU. Ada dua protokol yang digunakan dalam contoh program ini, protokol untuk mengendalikan atau mengatur mode kerja LED dan

protokol untuk membaca mode kerja LED saat ini. Protokol untuk mengatur mode LED ditentukan sebagai berikut:

LED#NomorLED#ModeLED#StatusLED#KecepatanBerkedip[CR][LF]

- 'LED' adalah perintah untuk mengatur mode kerja.
- NomorLED, bernilai 1 – 5, menentukan LED mana yang akan diatur
- ModeLED, untuk mengatur mode LED, 1 untuk mode berkedip dan 0 untuk mode ON/OFF.
- StatusLED, untuk menyalakan atau mematikan LED ketika LED bekerja di mode ON/OFF, 1 untuk menyalakan dan 0 untuk mematikan
- Kecepatanberkedip, untuk mengatur kecepatan berkedip dalam mili detik, ketika LED bekerja di mode berkedip.
- [CR][LF], karakter untuk mengakhiri 1 frame data [CR] = 0x0D dan [LF] = 0x0A.

Misal untuk mengatur LED1 di mode berkedip dengan kecepatan 500 mili detik, maka PC harus mengirim data sebagai berikut:

LED#1#1#0#500[CR][LF]

Atau mengatur LED 2 di mode ON/OFF dan sekaligus mematikannya:

LED#2#0#0#500[CR][LF]

Ketika MCU menerima perintah LED, MCU akan menjawab

LED Command OK[CR][LF]

Protokol kedua adalah protokol untuk membaca mode kerja LED:

RLM#NomorLED[CR][LF]

Ketika menerima perintah ini, MCU akan menjawab dengan format:

RLM#1#KecepatanBerkedip[CR][LF]

Ketika LED di mode berkedip, dan

RLM#0#StatusLED

Ketika LED bekerja di mode ON/OFF. Misal ketika menerima perintah untuk membaca status LED2 yang sedang berkedip dengan kecepatan 500 milidetik, MCU akan menjawab dengan

RLM#2#500

Atau ketika membaca status LED 3 yang bekerja di mode ON/OFF dan status LED sedang menyala:

RLM#0#1

Protokol yang didefinisikan di atas menggunakan teknik *delimited text/string* di mana beberapa data dalam bentuk teks disatukan dan dipisahkan oleh sebuah karakter yaitu '#'. Oleh karena itu diperlukan sebuah teknik untuk memisahkan data-data tersebut sesuai dengan definisi protokol. Teknik ini dinamakan dengan teknik *data parsing*. Dalam contoh program ini, teknik parsing ada di fungsi `parsing_delimiter_string`.

Listing Program 6.32 Fungsi Parsing Data

```
uint16_t parsing_delimiter_string(uint8_t *storebuffer, uint8_t
*databuffer, uint8_t delimiter, uint16_t startpos, uint16_t len){
    uint16_t i,x;
    x=0;
    for(i=startpos; i<startpos+len;i++){
        if((databuffer[i]==0)|| (databuffer[i]==delimiter))
            break;
        storebuffer[x++]=databuffer[i];
    }
    storebuffer[x++]=0;

    return i+1;
}
```

Fungsi `parsing_delimiter_string` ini memerlukan beberapa parameter:

- `uint8_t *strorebuffer`, sebuah pointer di mana data hasil parsing akan disimpan.
- `uint8_t *databuffer`, pointer yang menyimpan data yang akan diparsing
- `uint8_t delimiter`, karakter yang menjadi pemisah data
- `uint16_t startpos`, posisi awal proses parsing data dimulai
- `uint8_t len`, panjang dari data yang akan diparsing.

Nilai atau hasil dari fungsi ini adalah posisi untuk urutan data berikutnya. Fungsi akan membaca setiap byte dari databuffer dari posisi startpos dan menyimpannya di storebuffer, jika bertemu dengan karakter delimiter atau sudah membaca sampai panjang data (len), fungsi akan berhenti dan menghasilkan posisi pembacaan terakhir.

Ketika program utama menerima sinyal dari interupsi serial, nilai pcframe=1, program akan memanggil fungsi pc_link untuk menerjemahkan data yang dikirim oleh PC. Fungsi pc_link inilah yang menggunakan fungsi parsing data.

Listing Program 6.33 Fungsi pc_link

```
void pc_link(void){
    uint8_t cmd[4];
    uint8_t commanddata[16];
    uint16_t i,j;

    //Get command
    i= parsing_delimiter_string(cmd,pcbuffer,'#',0,pcbucounter);

    //Get command data
    memset(commanddata, '\0', 16);
    for(j=i;j<pcbucounter;j++){
        commanddata[j-4]=pcbuffer[j];
    }

    //Check command
    if(memcmp(cmd, "LED", 3)==0){
        set_led_mode(commanddata);
    }

    if(memcmp(cmd, "RLM", 3)==0){
        read_led_mode(commanddata);
    }

    memset(pcbuffer, '\0', sizeof(pcbuffer));
    pcbucounter=0;
    pcframe=0;
}
```

Fungsi pc_link ini hanya membaca perintah apa yang dikirim dari PC, LED atau RLM. Ketika ketemu perintah yang sesuai, pc_link akan memanggil fungsi yang bersangkutan. Misal ketika menerima perintah LED, pc_link akan memanggil fungsi set_led_mode.

Listing Program 6.34 Fungsi set mode LED

```
void set_led_mode(uint8_t* commanddata){
    uint8_t ledindex;
    uint8_t ledmode;
    uint8_t ledstatus;
    uint16_t blinkrate;
    uint8_t bfr[8];
    uint16_t i;
```

```
//Command Format
//LedIndex#LedMode#LedStatus#BlinkRate

//Get LED Index
i=parsing_delimiter_string(bfr, commanddata,'#',0,
strlen((char*)commanddata));
ledindex=atoi((char*)bfr);

//LED Mode
i=parsing_delimiter_string(bfr, commanddata,'#',i,
strlen((char*)commanddata));
ledmode=atoi((char*)bfr);

//LED Status
i=parsing_delimiter_string(bfr,
commanddata,'#',i,strlen((char*)commanddata));
ledstatus=atoi((char*)bfr);

//Blink Rate
i=parsing_delimiter_string(bfr,
commanddata,'#',i,strlen((char*)commanddata));
blinkrate=atoi((char*)bfr);

Set_LED_Operation(ledindex-1, ledmode, blinkrate,
ledstatus);

printf("LED Command OK\r\n");

}
```

Protokol LED mempunyai urutan data sebagai berikut:

NomorLED#ModeLED#StatusLED#KecepatanBerkedip

Terdapat 4 *field* data, oleh karena itu diperlukan 4 kali pemanggilan fungsi parsing data. Oleh karena data-data dikirim dalam bentuk ASCII (string), maka setelah dipisahkan, data tersebut langsung diubah sesuai dengan tipe variabel yang dituju.

Fungsi parsing data ini, tidak hanya bisa digunakan dalam komunikasi serial, tetapi bisa juga disimpan untuk, misalnya dalam menyimpan dan membaca data ke sebuah file di mikro SD. Data disimpan dalam teks delimiter, sehingga fungsi ini bisa digunakan untuk menterjemahkan isi file tersebut.

6.3 BERMAIN-MAIN DENGAN TIMER

Fungsi utama sebuah timer tentu saja adalah sebagai sumber pewaktuan (*time base*), karena pada dasarnya sebuah timer adalah pencacah yang akan melakukan cacahan dengan kecepatan clock tertentu, waktu pencacahan itulah yang dijadikan sebagai pewaktuan. Selain itu timer mempunyai sebuah keluaran yang akan di-set atau di-clear saat proses cacahan berakhir. Output ini bisa diprogram untuk membentuk sebuah sinyal PWM yang banyak digunakan untuk mengendalikan kecepatan sebuah motor listrik atau untuk mengatur kecerahan sebuah LED.

6.3.1 TIMER SEBAGAI SUMBER PEWAKTUAN

Hampir semua contoh program yang telah dibuat menggunakan atau memanggil fungsi HAL_Delay. Fungsi yang berfungsi untuk memberi waktu jeda dalam milidetik sebelum meng-eksekusi baris perintah selanjutnya. Fungsi delay pada dasarnya membiarkan CPU tidak melakukan apa-apa jeda waktu tertentu, biasanya fungsi delay isinya hanya membuat CPU untuk melakukan sebuah hitungan. Agar hitungan tersebut berlangsung sesuai yang diinginkan, maka dipakailah sebuah timer.

Listing Program 6.35 Fungsi HAL_Delay

```
_weak void HAL_Delay(__IO uint32_t Delay)
{
    uint32_t tickstart = HAL_GetTick();
    uint32_t wait = Delay;

    /* Add a period to guarantee minimum wait */
    if (wait < HAL_MAX_DELAY)
    {
        wait++;
    }

    while((HAL_GetTick() - tickstart) < wait)
    {
    }
}
```

STM32CubeMX sendiri secara *default* akan menggunakan Timer SysTick untuk fungsi delay. Timer SysTick difungsikan sebagai timer 1 milidetik, akan memberikan interupsi setiap 1 milidetik. Fungsi layanan interupsi timer SysTick hanya melakukan penambahan sebuah variabel uwTick, artinya variabel ini akan diperbarui setiap 1 mili detik.

Listing Program 6.36 Fungsi SysTick Handler

```
void SysTick_Handler(void)
{
    /* USER CODE BEGIN SysTick_IRQn 0 */

    /* USER CODE END SysTick_IRQn 0 */
    HAL_IncTick();
    HAL_SYSTICK_IRQHandler();
    /* USER CODE BEGIN SysTick_IRQn 1 */

    /* USER CODE END SysTick_IRQn 1 */
}

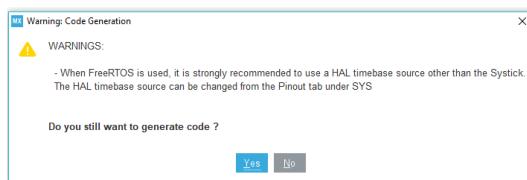
__weak void HAL_IncTick(void)
{
    uwTick++;
}

void HAL_SYSTICK_IRQHandler(void)
{
    HAL_SYSTICK_Callback();

    /**
     * @brief SYSTICK callback.
     * @retval None
     */
    __weak void HAL_SYSTICK_Callback(void)
{
    /* NOTE : This function Should not be modified, when the callback
is needed,
           the HAL_SYSTICK_Callback could be implemented in the
user file
    */
}
```

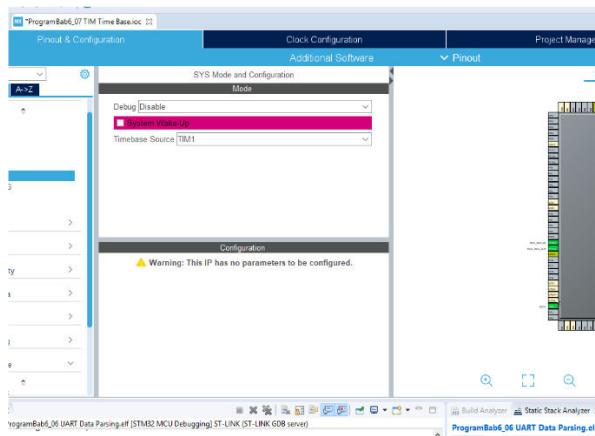
Jadi dalam fungsi HAL_Delay, pertama ambil nilai uwTick yang sekarang melalui fungsi HAL_GetTick (disimpan di variabel tickstart), kemudian ambil nilai uwTick terbaru, tunggu sampai selisih uwTick terbaru dengan tickstart sama dengan nilai Delay yang diinginkan.

Pertanyaannya bisakah timer SysTick digantikan oleh timer lain yang dimiliki oleh STM32F207? Tentu saja sangat bisa. Apalagi kalau program yang dibuat menggunakan sebuah kernel RTOS (FreeRTOS), STM32CubeIDE sendiri akan menyarankan agar tidak menggunakan timer SysTick sebagai timer pewaktuan untuk HAL. Karena pada saat RTOS digunakan, timer SysTick akan digunakan oleh kernel RTOS ketika menjalankan *scheduler*. Pembahasan tentang RTOS akan dilakukan di bab tersendiri.



Gambar 6.51 Rekomendasi STM32CubeMX

Untuk menggunakan timer lain yang dimiliki oleh STM32F207 sebagai sumber pewaktuan menggantikan timer SysTick, dari tab Pinout & Configuration menu System Core/SYS pilih sumber pewaktuan (*Timebase Source*) dan pilih salah satu timer yang tersedia. Kemudian generate code seperti seperti biasa, setelah melakukan pengaturan proyek, mengatur periperal dan sebagainya.



Gambar 6.52 Pemilihan Timer sebagai sumber pewaktuan

Salah satu yang membedakan adalah STM32CubeMX akan menghasilkan fungsi `HAL_InitTick()` dengan menkonfigurasikan timer yang dipilih menjadi sebuah timer yang akan membangkitkan interupsi setiap 1 milidetik. Secara default fungsi `HAL_InitTick` merupakan fungsi `_weak` dan mengarah ke timer SysTick. Ketika timer SysTick yang dipakai fungsi `_weak` ini yang akan dipanggil oleh compiler.

Listing Program 6.37 Fungsi `HAL_InitTick` Default

```
_weak HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
    /*Configure the SysTick to have interrupt in 1ms time basis*/
    HAL_SYSTICK_Config(SystemCoreClock/1000U);

    /*Configure the SysTick IRQ priority */
}
```

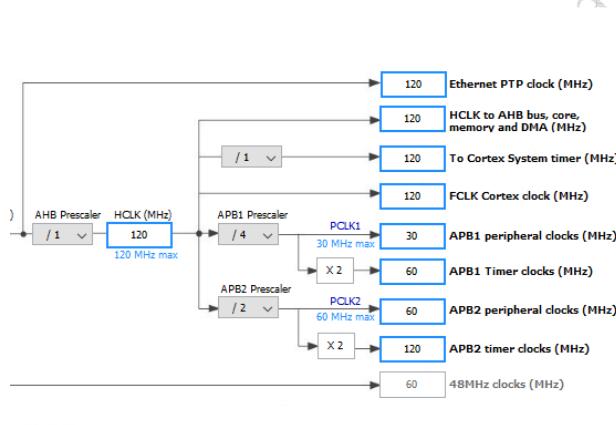
```

    HAL_NVIC_SetPriority(SysTick_IRQn, TickPriority ,0U);

    /* Return function status */
    return HAL_OK;
}

```

Misal ketika TIM1 yang digunakan, maka fungsi HAL_InitTick (tanpa __weak) akan mengatur TIM1 sebagai timer 1 milidetik. Clock TIM1 di-set di 1 MHz, clock ini berasal dari bus APB2 yang maksimal frekuensinya adalah 120 MHz (khusus untuk clock timer yang terhubung ke bus APB2, untuk periperal clock APB2 maksimal 60 MHz). Oleh karena itu nilai pra-skalar untuk TIM2 harus bernilai 120 – 1. Nilai pra-skalar ini berasal dari variabel uwPrescalerValue yang berasal dari pembagian variabel uwTimclock. Variabel uwTimclock sendiri merupakan $2 \times$ frekuensi PCLK2 (bus APB2). Untuk mendapatkan interupsi 1 milidetik (frekuensi 1KHz), TIM1 harus overflow setelah 1000 kali cacahan. Oleh karena itu nilai periode akan bernilai $(1000000/1000) - 1$ ($1\text{MHz}/1000 = 1\text{KHz}$).



Gambar 6.53 Clock Bus AHB dan APB STM32F207

Untuk mencari parameter timer bisa dilakukan dengan persamaan berikut. Nilai pra-skalar dihitung sebagai berikut:

$$\text{Pra-Skalar} = (\text{Frekuensi APB Timer}/\text{Frekuensi Clock Timer}) - 1$$

Sebagai contoh, jika Frekuensi APB Timer = 120MHz dan Frekuensi Clock Timer yang diinginkan 1 MHz, maka:

$$\text{Pra-Skalar} = (120.000.000/1000.000) - 1$$

$$= 119$$

Nilai Pra-Skalar ini akan disimpan di register TIMx_PSC.

Dan agar timer berinterupsi setiap 1 mili detik atau 1 KHz dalam frekuensi, maka nilai perioda dihitung dengan persamaan berikut:

$$\text{Perioda} = (\text{Frekuensi Clock Timer}/\text{Frekuensi Timer})$$

$$= (1.000.000/1000) - 1$$

$$= 999$$

Nilai Perioda ini akan disimpan di register TIMx_ARR.

Listing Program 6.38 Fungsi HAL_InitTick dengan Timer

```
HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
    RCC_ClkInitTypeDef      clkconfig;
    uint32_t                 uwTimclock = 0;
    uint32_t                 uwPrescalerValue = 0;
    uint32_t                 pFLatency;

    /*Configure the TIM1 IRQ priority */
    HAL_NVIC_SetPriority(TIM1_UP_TIM10_IRQn, TickPriority ,0);

    /* Enable the TIM1 global Interrupt */
    HAL_NVIC_EnableIRQ(TIM1_UP_TIM10_IRQn);

    /* Enable TIM1 clock */
    __HAL_RCC_TIM1_CLK_ENABLE();

    /* Get clock configuration */
    HAL_RCC_GetClockConfig(&clkconfig, &pFLatency);

    /* Compute TIM1 clock */
    uwTimclock = 2*HAL_RCC_GetPCLK2Freq();

    /* Compute the prescaler value to have TIM1 counter clock equal to
    1MHz */
    uwPrescalerValue = (uint32_t) ((uwTimclock / 1000000) - 1);

    /* Initialize TIM1 */
    htim1.Instance = TIM1;

    /* Initialize TIMx peripheral as follow:
    + Period = [(TIM1CLK/1000) - 1]. to have a (1/1000) s time base.
    + Prescaler = (uwTimclock/1000000 - 1) to have a 1MHz counter
    clock.
    + ClockDivision = 0
    + Counter direction = Up
    */
    htim1.Init.Period = (1000000 / 1000) - 1;
```

```

htim1.Init.Prescaler = uwPrescalerValue;
htim1.Init.ClockDivision = 0;
htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
if(HAL_TIM_Base_Init(&htim1) == HAL_OK)
{
    /* Start the TIM time Base generation in interrupt mode */
    return HAL_TIM_Base_Start_IT(&htim1);
}
/* Return function status */
return HAL_ERROR;
}

```

Hal lain yang membedakan ketika TIM1 digunakan sebagai sumber pewaktuan adalah fungsi HAL_IncTick akan dipanggil ketika terjadi interupsi TIM1 di fungsi layanan interupsi TIM1 melalui fungsi *callback*.

Listing Program 6.39 Fungsi Callback TIM1

```

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    /* USER CODE BEGIN Callback 0 */

    /* USER CODE END Callback 0 */
    if (htim->Instance == TIM1) {
        HAL_IncTick();
    }
    /* USER CODE BEGIN Callback 1 */

    /* USER CODE END Callback 1 */
}

```

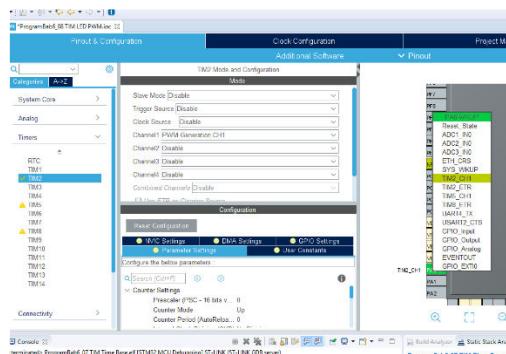
6.3.2 TIMER SEBAGAI PEMBANGKIT SINYAL PWM

Timer mempunyai keluaran (*Output Compare*) yang dipetakan juga ke salah satu pin GPIO. Keluaran ini bisa diatur (set atau reset) sesuai dengan overflow atau underflow dari pencacah timer (perioda). Dengan mengatur periода ini maka waktu set atau reset keluaran timer bisa diatur, dan pin GPIO yang dipetakan ke keluaran timer akan mengeluarkan sinyal yang dinamakan dengan sinyal PWM. Sinyal PWM bisa digunakan untuk pengendalian motor listrik di dunia industri atau robotik.

Dalam aplikasi sederhana sinyal PWM bisa digunakan untuk mengatur kecerahan sebuah LED seperti yang akan dijelaskan di contoh proyek berikut. Dalam proyek blink sebenarnya LED sudah diberikan sinyal PWM dengan siklus kerja 50%, namun karena frekuensinya sangat rendah, maka LED masih terlihat berkedip. Jika frekuensinya cukup

tinggi, dengan LED tidak akan terlihat berkedip tapi akan terlihat menyala dengan tingkat kecerahan 50% (redup). Artinya dengan mengatur siklus kerja sinyal PWM, maka LED bisa diatur tingkat kecerahannya (terang redupnya) atau fungsi dimmer.

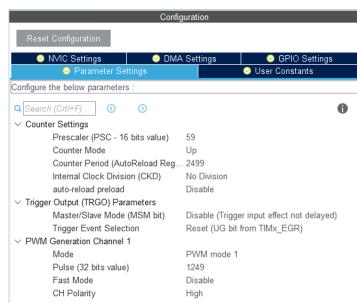
Pin PA0 yang digunakan untuk proyek blink mempunyai fungsi alternatif sebagai keluaran timer, yaitu TIM2_CH1 dan TIM5_CH1. Dengan memfungsikan PA0 sebagai keluaran timer, misal TIM2_CH1, PA0 bisa digunakan untuk mengatur kecerahan LED. TIM2 diatur sebagai pembangkit PWM di channel 1 (*PWM Generation CH1*).



Gambar 6.54 Fungsi Alternatif PA0

Frekuensi sinyal PWM diatur agar cukup cepat sehingga tidak akan terlihat efek kedipnya dan juga tidak terlalu tinggi sehingga LED bisa mengikuti kecepatan sinyal PWM, dalam contoh program ini frekuensi diatur di 400 Hz. Sedangkan siklus kerja disesuaikan sehingga LED akan terlihat menyala dari redup perlahan menjadi terang dan kemudian perlahan meredup kembali secara berulang-ulang.

TIM2 terhubung dengan bus APB1 dan akan mendapat clock sebesar 60 MHz pada clock CPU 120 MHz. Dengan menggunakan persamaan-persamaan dari contoh program sebelumnya, maka Pra-skalar diatur di 59 (60-1) sehingga TIM2 akan mendapat clock 1 MHz. Agar TIM2 menghasilkan frekuensi 400Hz, periода diatur di 2500-1 (2499). Siklus kerja, yang pada dasarnya nilai persentasi dari perioda, diatur dengan parameter *Pulse*. Nilai pulsa akan menentukan menentukan berapa lama output timer berada di kondisi ON. Nilai pulsa tidak boleh melebihi nilai perioda. Sedangkan kondisi logika keluaran timer saat ON ditentukan oleh *CH Polarity* yang bisa bernilai *High* atau *Low*. Nilai Pulsa ini di simpan di register *TIMx_CCR3*.



Gambar 6.55 Pengaturan TIM2

Listing Program 6.40 Inisialisasi TIM2

```
void MX_TIM2_Init(void)
{
    TIM_MasterConfigTypeDef sMasterConfig;
    TIM_OC_InitTypeDef sConfigOC;

    htim2.Instance = TIM2;
    htim2.Init.Prescaler = 59;
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim2.Init.Period = 2499;
    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }

    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig)
!= HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }

    sConfigOC.OCMode = TIM_OCMODE_PWM1;
    sConfigOC.Pulse = 1249;
    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1)
!= HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }

    HAL_TIM_MspPostInit(&htim2);
```

```
}
```

Setelah menginisialisasi TIM2, program akan mengatur PWM dengan menaikkan nilai siklus kerja 5% setiap 200 mili detik, setelah mencapai 100%, siklus kerja akan diturunkan kembali 5% setiap 200 mili detik. Dan ketika siklus kerja telah turun sampai 5%, siklus kerja kembali akan dinaikan 5% setiap 200 mili detik. Variabel mode menentukan apakah siklus kerja sedang naik atau turun. Ketika siklus kerja sedang naik, variabel addvalue akan bernilai 5 dan ketika sedang turun addvalue akan bernilai -5.

Listing Program 6.41 Program Utama

```
dutycycle=5;
addvalue=5;
mode=0;
sConfigOC.OCMode = TIM_OCMODE_PWM1;
sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */
    sConfigOC.Pulse = (2499 * dutycycle)/100;
    if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC,
        TIM_CHANNEL_1) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }

    if (HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1) != HAL_OK)
    {
        /* PWM Generation Error */
        Error_Handler();
    }

    dutycycle+=addvalue;
    if((!mode)&&(dutycycle==100)){
        mode=1;
        addvalue=-5;
    }
    else{
        if(dutycycle == 5){
            mode=0;
            addvalue=5;
        }
    }
}
```

```

    }

    HAL_Delay(200);

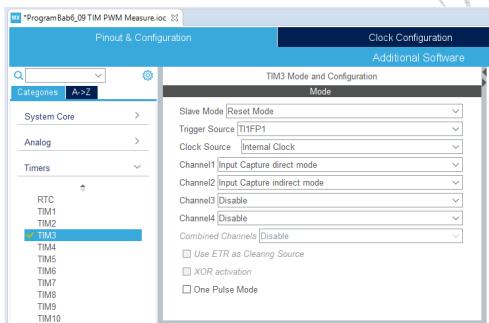
/* USER CODE BEGIN 3 */

}

```

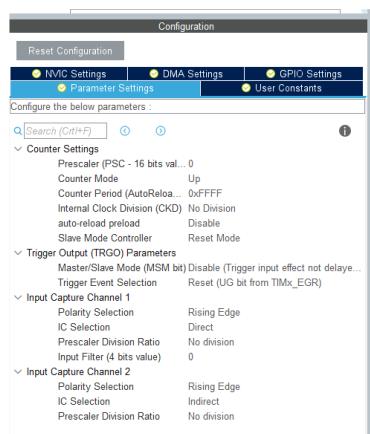
6.3.3 TIMER SEBAGAI PENGUKUR SINYAL

Selain mempunyai keluaran, timer juga mempunyai masukan yang juga dipetakan ke pin GPIO yang sama. Pin masukan timer ini bisa digunakan untuk pengukuran sinyal yang berasal dari luar, frekuensi maupun siklus kerjanya. Seperti telah dijelaskan di bab sebelumnya, untuk pengukuran frekuensi dan siklus kerja sebuah sinyal diperlukan 2 buah input capture yang dipetakan ke pin input timer yang sama. Kedua unit capture ini bekerja secara berlawanan, unit capture yang pertama mendeteksi sinyal saat transisi naik sedangkan unit capture yang lain mendeteksi saat transisi turun. Unit capture yang pertama akan mengukur periода sinyal (frekuensi) sedangkan unit capture yang lainnya perioda saat lebar pulsa (siklus kerja).



Gambar 6.56 Pengukuran Sinyal Melalui Input TIM3 CH1

Pada contoh program ini, digunakan TIM3 CH1 dan CH2 yang dihubungkan ke pin PA6 sebagai pin masukan timer TIM3. CH1 diatur untuk mode langsung (*direct*) artinya terkoneksi dengan PA6 (TIM3_CH1) sedangkan dan CH2 mode tak langsung yang artinya juga terkoneksi dengan PA6 (sumber pemicu dipilih TI1FP1). Mode slave diatur di mode reset dan sumber clock timer menggunakan clock internal. CH1 diatur untuk mendeteksi transisi sinyal naik dan CH2 diatur untuk mendeteksi transisi turun (lihat gambar 3.27 untuk ilustrasi). Dan interupsi TIM3 diaktifkan.



Gambar 6.57 Pengaturan TIM3

Pada mode input ini, pencacah akan melakukan cacahan dengan sumber clock internal (mode reset). Pada saat pertama kali ada transisi naik di sinyal masukan, unit capture CH1 dan CH2 akan mulai pencacahan. Ketika ada transisi turun, CH2 akan menyimpan nilai perioda pulsa dari sinyal yang diukur (siklus kerja) sedangkan saat transisi naik berikutnya, CH1 yang akan menyimpan perioda sinyal yang diukur (frekuensi). Saat kondisi *capture* di kedua channel, timer akan memberikan interupsi, berarti akan terjadi 2 kali interupsi setiap satu siklus sinyal yang diukur, yaitu dari CH2 (saat transisi turun) dan CH1 (saat transisi naik). Di program ini, cukup saat interupsi CH1 saja yang ditangani, karena pada saat itu, CH2 masih tetap menyimpan nilai siklus kerja.

Listing Program 6.42 Fungsi Layanan Interupsi Capture Timer

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim){
    if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)
    {
        /* Get the Input Capture value */
        uwIC2Value = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);

        if (uwIC2Value != 0)
        {
            /* Duty cycle computation */
            uwDutyCycle = ((HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2))
* 100) / uwIC2Value;

            /* uwFrequency computation
            TIM2 counter clock = (RCC_Clocks.HCLK_Frequency)/2 */
            uwFrequency = (HAL_RCC_GetHCLKFreq()) / 2 / uwIC2Value;
        }
    }
}
```

```
    else
    {
        uwDutyCycle = 0;
        uwFrequency = 0;
    }
}
```

CH1 dan CH2 menyimpan data sinyal dalam bentuk hasil cacahan. Untuk mendapatkan siklus kerja dalam persentase tinggal dihitung saja nilai persentasi cacahan di CH2 terhadap cacahan di CH1, lebar pulsa terhadap perioda sinyal. Maka siklus kerja bisa dihitung dengan rumus:

$$\text{Siklus Kerja} = (\text{Cacahan CH2}/\text{Cacahan CH1}) \times 100\%$$

Sedangkan nilai frekuensi dihitung dengan persamaan:

$$\text{Frekuensi} = \text{Frekuensi Clock Timer} / \text{Cacahan CH1}$$

Nilai cacahan (capture) di CH1 dan CH2 diperoleh dengan fungsi HAL_TIM_ReadCapturedValue.

Ada persyaratan agar pengukuran bisa dilakukan. Prinsip pengukuran ini pada dasarnya membandingkan sinyal yang diukur dengan pencacahan yang dilakukan oleh timer. Oleh karena itu, perioda sinyal yang diukur tidak boleh terlalu lama agar unit capture timer tidak mengalami overflow sebelum satu siklus. Dengan kata lain, frekuensi tidak boleh terlalu rendah. Selain itu, perioda sinyal juga tidak boleh terlalu cepat yang akan mengakibatkan ketika terjadi satu siklus, timer belum sempat melakukan cacahan. Dengan kata lain frekuensi tidak boleh terlalu tinggi.

Frekuensi minimum yang dapat diukur oleh contoh program ini dihitung dengan persamaan:

$$\text{Frekuensi Minimum} = \text{Frekuensi Clock Timer} / \text{Nilai Capture Maksimum}$$

Di program ini, TIM3 diprogram untuk mempunyai clock 60 MHz dan nilai capture maksimum adalah 65535 (16 bit), maka frekuensi minimum adalah:

$$\text{Frekuensi Minimum} = 60 \text{ MHz} / 65535$$

$$= 915,5 \text{ Hz}$$

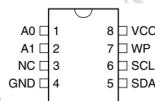
Untuk mengukur frekuensi lebih rendah, bisa dilakukan dengan menurunkan frekuensi timer pembanding (TIM3) atau bisa juga dengan

menggunakan timer 32 bit (TIM2 atau TIM5). Selain itu penggunaan timer 32 bit bisa juga meningkatkan akurasi pengukuran.

6.4 BERMAIN-MAIN DENGAN I2C

Pada program parsing data yang mengatur mode nyala 5 buah LED, setelah data pengaturan LED dikirim dari PC akan disimpan di RAM internal mikrokontroler. Artinya ketika mikrokontroler di-reset atau catu daya dimatikan, settingan akan kembali ke awal (default). Untuk mengatasi hal ini data pengaturan harus disimpan di memori non-volatile, yang akan tetap menyimpan data walau tidak ada catu daya. Contoh dari memori ini adalah EEPROM (*Electrically Erase Programmable Read Only Memory*). Salah satu EEPROM yang sering digunakan adalah EEPROM dengan antarmuka I2C, misalnya 24C256.

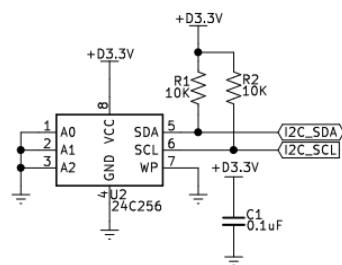
24C256 merupakan EEPROM I2C dengan kapasitas 262144 (256 kbit) bit atau 32768 byte (32 KB). EEPROM ini biasa dibuat dengan kemasan PDIP (*Plastic Dual In-line Package*), TSSOP (*Thin Shrink Small Outline Package*), SOIC (*Small Outline Integrated Circuit*), MAP (*MSOP Array Package*) bahkan BGA (*Ball Grid Array*) yang semuanya mempunyai 8 pin. IC memori ini bisa bekerja di tegangan 2.7V (2.7V – 5.5V) atau 1.8V (1.8 – 3.6V) dan dilengkapi dengan fitur proteksi terhadap penulisan dengan cara menghubungkan pin WP (*Write Protect*) ke VCC (logika tinggi).



Gambar 6.58 Konfigurasi Pin 24C256 SOIC

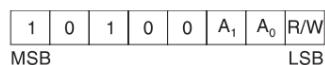
EEPROM bisa dihubungkan dengan port I2C mana saja yang dimiliki oleh STM32F207. Resistor 10K harus selalu dipasang di pin SDA dan SCL sebagai pull-up mengikuti aturan I2C. Pin WP (*Write Protect*) dihubungkan ke ground untuk operasi normal, apabila diinginkan untuk proteksi terhadap penulisan bisa dihubungkan ke VCC (3.3V). Pin WP ini mempunyai pull-down internal, sehingga apabila mengambang akan tetap terbaca sebagai logika 0. Pin alamat (A0 dan A1) bisa dihubungkan ke GND atau VCC. Dengan mengatur kombinasi pin alamat 4 buah 24C256 bisa dihubungkan paralel dalam 1 bus I2C. Atau bisa juga dihubungkan paralel dengan perangkat I2C lain selama alamatnya

berbeda. Dan kapasitor 0.1uF disarankan sebagai filter catu daya untuk setiap IC, dan di PCB ditempatkan sedekat mungkin dengan IC tersebut.



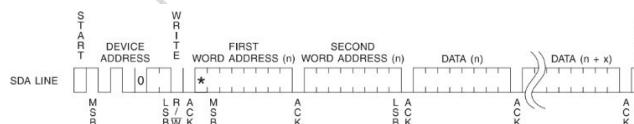
Gambar 6.59 Skematik Antarmuka AT24C256

Sesuai dengan standar I2C, 24C256 mempunyai 8 bit alamat (*device address*) dengan 2 bit alamat LSB diatur oleh secara eksternal melalui pin A0 dan A1, sehingga 4 buah 24C256 bisa diparalel dalam satu bus I2C. Dan bit ke-0 (LSB) merupakan bit yang mengatur operasi baca atau tulis, 0 untuk operasi tulis dan 1 untuk operasi baca. Misal ketika pin A0 dan A1 dihubungkan ke logika 0 (GND), 24C256 akan mempunyai alamat 0xA0 saat menulis ke memori dan alamat 0xA1 saat membaca dari memori.



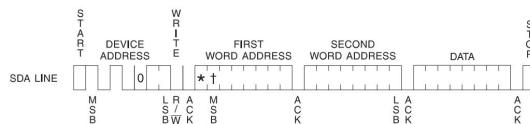
Gambar 6.60 Alamat 24C256

Memorinya sendiri dibagi ke dalam 512 page yang berisi masing-masing 64 byte. Operasi baca tulis bisa dilakukan per page atau per byte. Memori dialamati dengan 15 bit. Alamat ini hanya sekali dikirim ketika operasi page. EEPROM akan akan menerima (operasi tulis) atau mengirim data (operasi baca) secara berurutan selama belum menerima bit STOP.



Gambar 6.61 Operasi Tulis Per Page

Pada operasi byte, bit STOP harus dikirim setelah melakukan penulisan atau pembacaan setiap byte. Untuk mengulangi operasi byte berikutnya harus dimulai dari bit START.



Gambar 6.62 Penulisan Per Byte

Pustaka HAL telah menyediakan fungsi untuk membaca atau menulis ke EEPROM I2C baik operasi byte maupun operasi page. Untuk membaca data dari EEPROM, HAL menyediakan fungsi

Listing Program 6.43 Fungsi HAL Membaca Data dari EEPROM

```
HAL_StatusTypeDef HAL_I2C_Mem_Read(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData, uint16_t Size, uint32_t Timeout)
```

Sedangkan untuk menulis ke EEPROM digunakan fungsi

Listing Program 6.44 Fungsi HAL Menulis ke EEPROM

```
HAL_StatusTypeDef HAL_I2C_Mem_Write(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

Fungsi-fungsi ini membutuhkan parameter-parameter pointer dari port I2C yang dipakai (*hi2c), alamat I2C (DevAddress), alamat memori yang akan diakses (MemAddress), ukuran dari alamat (MemAddSize), 24C256 membutuhkan 16 bit alamat, EEPROM lain misal 24C08 membutuhkan 8 bit alamat. Parameter selanjutnya adalah pointer terhadap data (*pData), banyaknya data (Size) dan timeout (Timeout) yaitu waktu tunggu yang diperlukan apabila perangkat I2C tidak merespon.

Untuk mempermudah, bisa dibuat fungsi membaca dan menulis ke EEPROM yang akan memanggil kedua fungsi tersebut:

Listing Program 6.45 Fungsi Membaca dan Menulis EEPROM

```
uint8_t EEPROM_Read(uint16_t Addr,uint8_t *buf,uint16_t num){
    uint8_t temp;

    temp = HAL_I2C_Mem_Read(&hi2c2, EEPROM_ADDR,
    Addr,I2C_MEMADD_SIZE_16BIT, buf, num, 0xFFFF);
    return temp;
}

uint8_t EEPROM_Write(uint16_t Addr,uint8_t *buf,uint16_t num){
    uint8_t temp;
    temp = HAL_I2C_Mem_Write(&hi2c2, EEPROM_ADDR,
    Addr,I2C_MEMADD_SIZE_16BIT, buf, num, 0xFFFF);
    HAL_Delay(10);
}
```

```
        return temp;
}
```

Untuk tipe EEPROM yang berbeda, misal 24C08 yang membutuhkan lebar alamat 8 bit, tinggal mengganti MEMADD_SIZE_16BIT menjadi MEMADD_SIZE_8BIT.

Perhatikan kembali program tentang parsing data UART. Program tersebut akan mengatur mode kerja LED melalui port serial. Mode kerja LED akan dikirim dengan perintah tertentu. Namun ketika mikrokontroler di-reset atau catu daya dihilangkan maka mode kerja akan kembali ke mode defaultnya. Untuk itu, seperti telah dijelaskan di awal sub-bab ini, mode kerja perlu disimpan di EEPROM.

Mode kerja LED didefinisikan dengan struktur data sebagai berikut:

Listing Program 6.46 Struktur Data Mode Kerja LED

```
typedef struct{

    uint8_t  status;
    uint32_t timer;
    uint8_t  mode;
    uint16_t blinkrate;

}LEDTypeDef;
```

Dari struktur data tersebut, data yang harus disimpan di EEPROM adalah status (1 byte), mode (1 byte) dan blinkrate (2 byte). Sehingga setiap LED membutuhkan 4 byte untuk menyimpan mode operasinya. Variabel timer tidak perlu untuk disimpan di EEPROM karena variabel ini akan selalu diperbarui dengan mengambil nilai dari fungsi HAL_GetTick.

Ketika menggunakan EEPROM, perlu diperhatikan bahwa EEPROM akan berisi 0xFF ketika keluar dari pabrik. Untuk itu diperlukan sebuah cara untuk mengetahui bahwa EEPROM yang dipakai adalah baru atau belum menyimpan data pengaturan sebelumnya. Sehingga apabila data EEPROM terbaca 0xFF atau format datanya tidak sesuai dengan format data yang telah ditentukan, bisa bekerja di mode default sehingga alat yang dibuat tetap bekerja sesuai dengan rancangan. Cara yang bisa dipakai adalah dengan menggunakan *checksum* atau CRC. CRC bisa menggunakan 8 bit (1 byte) atau 16 bit (2 byte). CRC juga sering digunakan dalam komunikasi, untuk mengecek apakah data yang diterima terjadi error atau tidak. Dalam contoh program kali ini akan digunakan CRC 8 bit.

Data-data pengaturan LED tersebut akan dihitung CRC-nya. Setiap LED mempunyai CRC masing-masing dan CRC tersebut juga harus disimpan dalam EEPROM. Sehingga memori yang dibutuhkan menjadi 5 byte untuk setiap LED. Setiap kali reset, program akan membaca data pengaturan dan CRCnya. Data pengaturan kemudian akan dihitung CRCnya dan dibandingkan dengan nilai CRC yang terbaca. Jika nilainya sama, maka data pengaturan valid, jika tidak berarti belum ada pengaturan sebelumnya sehingga akan digunakan data pengaturan default. Begitu juga ketika terjadi pengaturan mode kerja dari komputer, data pengaturan yang dikirim akan dihitung CRCnya dan akan disimpan di EEPROM.

CRC bisa dihitung dengan fungsi sederhana berikut:

Listing Program 6.47 Fungsi Penghitungan CRC

```
uint8_t CalculateCRC8(uint8_t* buffer, int length)
{
    uint8_t crc8,tmp,j,k;

    crc8 =0;
    if (!length)
        crc8= 0xFF;
    else
    {
        while (length--)
    {
        tmp = (*buffer++);
        for (k=0;k<7;k++)
        {
            j = 1 & (tmp ^ crc8);
            crc8 = (crc8/2) & 0xFF;
            tmp = (tmp/2) & 0xFF;
            if (j!=0)
                crc8 ^= 0x8C;
        }
    }
    return (crc8);
}
```

Setiap kali program restart, sebelum mengeksekusi fungsi kerja LED (LED_Operation()), data pengaturan LED tersebut dibaca dari EEPROM dan dihitung CRC-nya, jika CRC-nya valid maka data pengaturan akan mengambil data yang tersimpan di EEPROM, jika tidak akan menggunakan mode pengaturan default. Berikut contoh untuk membaca

data pengaturan LED ke-1. Data pengaturan akan berada di byte ke-0 sampai byte ke-3 (4 byte) sedangkan data CRC berada di byte ke-4.

Listing Program 6.48 Membaca Data Pengaturan dari EEPROM

```
address=0x0;
EEPROM_Read(address,bfr,25);
if(bfr[4]==CalculateCRC8(bfr,4)){
    LED[0].status=bfr[0];

    LED[0].mode = bfr[1];

    LED[0].blinkrate = bfr[3];
    LED[0].blinkrate = LED[0].blinkrate << 8 | bfr[2];

}
else{
    LED[0].mode = LEDMODE_BLINK;
    LED[0].status = LEDSTATUS_OFF;
    LED[0].timer = HAL_GetTick();
    LED[0].blinkrate = 50;
}
```

Dalam contoh program ini, data pengaturan untuk LED1 disimpan di alamat 0. Oleh karena setiap LED membutuhkan 5 byte (termasuk CRC), maka alamat penyimpanan untuk LED selanjutnya adalah, dalam kelipatan 5, 0x05 (LED2), 0x0A (LED3), 0x0F (LED4), dan 0x14 (LED5).

Untuk penyimpanan data ke EEPROM, dilakukan setiap kali ada pengaturan melalui PC. Data pengaturan disimpan terlebih dahulu ke sebuah buffer 4 byte dan dihitung CRC-nya, kemudian disimpan ke EEPROM bersama nilai CRCnya.

Listing Program 6.49 Fungsi Menyimpan Data Pengaturan LED

```
void Write_LED_Setting(uint8_t LEDIndex){
    uint8_t bfr[10];
    uint16_t address;

    address = (LEDIndex)*5;
    bfr[0]=LED[LEDIndex].status;

    bfr[1]=LED[LEDIndex].mode;

    bfr[2] = LED[LEDIndex].blinkrate & 0xFF;
    bfr[3] = (LED[LEDIndex].blinkrate >>8) & 0xFF;

    bfr[4]=CalculateCRC8(bfr,4);
```

```
    EEPROM_Write(address,bfr,5);  
}
```

Seperti telah disebutkan, CRC juga bisa digunakan dalam sistem komunikasi. Pengiriman data mode kerja LED dari PC bisa menambahkan CRC di protokol komunikasi. Ketika menerima data pengaturan, CRC ini akan dicek untuk memastikan bahwa data yang diterima valid (tidak ada error). Protokol LED akan berubah menjadi

NomorLED#ModeLED#StatusLED#KecepatanBerkedip#CRC

Karena data dikirim dalam bentuk ASCII, CRC juga bisa diubah terlebih dahulu menjadi ASCII.

Bab 7

BERMAIN-MAIN DENGAN ADC/DAC

STM32F207 dilengkapi dengan periperal analog yaitu ADC dan DAC 12 bit. ADC bisa digunakan untuk membaca tegangan analog, misal pembacaan sensor dengan keluaran analog. Sedangkan DAC bisa digunakan untuk membangkitkan sinyal analog, misal untuk membangkitkan gelombang sinus, gelombang segitiga atau bahkan sinyal audio.

7.1 BERMAIN-MAIN DENGAN ADC

Dengan ADC yang dimilikinya, STM32F207 bisa digunakan untuk mengukur tegangan analog. Sensor-sensor yang dipakai di dunia industri banyak menggunakan keluaran analog baik dalam bentuk tegangan maupun arus. Di antara standar keluaran yang pakai adalah keluaran 0-5V dan 0-10V untuk tegangan dan 4-20mA untuk arus.

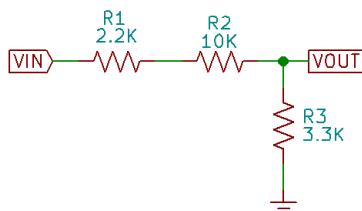
Tentu saja karena tegangan masukan untuk ADC dibatasi sampai 3.3V (DC), diperlukan rangkaian pengkondisi sinyal untuk menyesuaikan level tegangan yang ke level tegangan yang bisa diterima oleh ADC. Untuk keluaran arus harus diubah juga menjadi tegangan karena ADC hanya bisa mengukur tegangan. Selain itu, jika akan digunakan untuk mengukur tegangan AC, maka tegangan AC tersebut harus diberi tegangan offset agar menjadi tegangan DC.

Untuk pengukuran ADC yang lebih baik, tegangan analog (VDDA dan VSSA) bisa dipisahkan dari tegangan digital seperti telah dibahas di bab 4. Pada saat merancang PCB juga harus diperhatikan agar jalur-jalur

analog tidak bersilangan dengan jalur tegangan frekuensi tinggi, dan sedapat mungkin jalur analog tersebut dilindungi dengan area ground.

7.1.1 PENGUKURAN TEGANGAN DC

Oleh karena tegangan masukan ADC dibatasi sampai 3,3V (sesuai dengan tegangan referensi), maka untuk pengukuran tegangan lebih dari 3,3V diperlukan sebuah rangkaian pengkondisi sinyal (*signal conditioning*). Rangkaian pembagi tegangan merupakan rangkaian sederhana yang akan mengubah (membagi tegangan) yang lebih besar menjadi tegangan 3,3V atau kurang. Pada saat tegangan keluaran terukur 3,3V maka itulah tegangan maksimal yang masih bisa diukur oleh ADC STM32F207.



Gambar 7.1 Rangkaian Pembagi Tegangan

Gambar 7.1 menunjukkan rangkaian tegangan yang dipakai dalam contoh program sub bab ini. Dengan menerapkan hukum Ohm, tegangan Vout bisa dihitung dengan persamaan:

$$V_{out} = V_{in} \times \frac{R_3}{R_1 + R_2 + R_3}$$

Vout telah ditentukan di 3,3V begitu juga nilai semua resistor telah diketahui. Maka nilai Vin pada saat Vout bernilai 3,3V bisa dicari dengan persamaan:

$$V_{in} = \frac{V_{out} \times (R_1 + R_2 + R_3)}{R_3}$$

Dengan memasukan semua nilai yang diketahui didapat nilai Vin sebesar 15,5V, artinya dengan rangkaian pembagi tegangan tersebut, ADC internal STM32 bisa digunakan untuk mengukur tegangan DC sampai dengan 15,5V. Dengan demikian pembagi tegangan tersebut adalah 15,5/3,3. Nilai pembagi ini bisa juga digunakan untuk mencari nilai Vin:

$$V_{in} = V_{out} \times \frac{15,5}{3,3}$$

Lalu bagaimana mengubah nilai ADC (12 bit) menjadi nilai tegangan di Vin?

Pada saat tegangan input ADC mencapai 3,3V maka data ADC akan terbaca maksimal 4095 (12 bit). Sehingga hubungan antara tegangan input ADC dengan nilai ADC bisa dituliskan dengan persamaan:

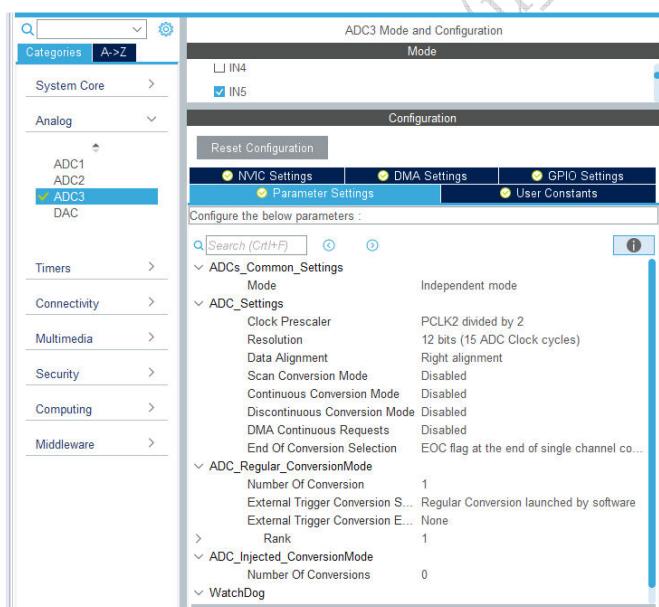
$$\text{Nilai ADC} = \frac{V_{in} \text{ ADC}}{3,3} \times 4095$$

Atau

$$V_{in} \text{ ADC} = \frac{\text{Nilai ADC}}{4095} \times 3,3$$

Oleh karena Vin ADC adalah Vout, maka

$$V_{in} = \frac{\text{Nilai ADC}}{4095} \times 15,5$$



Gambar 7.2 Pengaturan ADC

Contoh program menggunakan ADC3 channel 5 yang dipetakan ke GPIO pin PF7. ADC diprogram untuk bekerja di 12 bit dengan mode *polling*. Proses konversi dipicu melalui software dengan clock di set di 30 MHz. Waktu sampling diatur di 3 siklus dan waktu untuk konversi 12 bit data menjadi 12 siklus. Dengan clock 30 MHz, maka waktu konversi total adalah $(12+3)/30 = 0.5$ mikro detik.

Listing Program 7.1 Fungsi Inisialisasi ADC

```
void MX_ADC3_Init(void)
{
    ADC_ChannelConfTypeDef sConfig;

    /*Configure the global features of the ADC (Clock, Resolution,
    Data Alignment and number of conversion)
     */
    hadc3.Instance = ADC3;
    hadc3.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;
    hadc3.Init.Resolution = ADC_RESOLUTION_12B;
    hadc3.Init.ContinuousConvMode = DISABLE;
    hadc3.Init.DiscontinuousConvMode = DISABLE;
    hadc3.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    hadc3.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    hadc3.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc3.Init.NbrOfConversion = 1;
    hadc3.Init.DMAContinuousRequests = DISABLE;
    hadc3.Init.EOCSelection = DISABLE;
    if (HAL_ADC_Init(&hadc3) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }

    /*Configure for the selected ADC regular channel its
    corresponding rank in the sequencer and its sample time.
     */
    sConfig.Channel = ADC_CHANNEL_5;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
    if (HAL_ADC_ConfigChannel(&hadc3, &sConfig) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }
}
```

Oleh karena ADC diatur untuk terpicu secara software, maka untuk memulai proses konversi, program harus memanggil fungsi HAL_ADC_Start dan melakukan *polling* sampai proses konversi selesai melalui fungsi HAL_ADC_PollForConversion. Ketika status ADC terbaca

konversi telah selesai (*End of conversion*), nilai ADC dibaca dengan fungsi HAL_ADC_GetValue. Selanjutnya nilai ADC tersebut diubah menjadi nilai tegangan yang terukur dengan rumus yang telah disebutkan di atas dan dikirim ke port serial.

Untuk menghasilkan pembacaan yang stabil, pembacaan ADC mungkin perlu dilakukan berulang-ulang (misal 10 kali) kemudian diambil rata-ratanya baru dikirimkan ke port serial. Teknik pemfilteran mungkin juga diperlukan ketika masukan ADC adalah sensor analog yang sangat sensitif. Misalnya dengan menggunakan teknik filter Kalman dan lain-lain.

Listing Program 7.2 Program Utama Pembacaan ADC

```
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    //Start ADC Conversion
    if (HAL_ADC_Start(&hadc3)!=HAL_OK)
    {
        Error_Handler();
    }

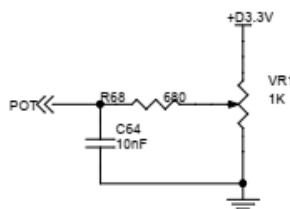
    //Poll ADC, wait until conversion finish
    HAL_ADC_PollForConversion(&hadc3, 10);
    if((HAL_ADC_GetState(&hadc3) & HAL_ADC_STATE_EOC_REG) ==
    HAL_ADC_STATE_EOC_REG)
    {
        //Read ADC Value
        uhADCxConvertedValue = HAL_ADC_GetValue(&hadc3);
    }

    //Convert to actual voltage
    Vin = (float) (uhADCxConvertedValue * 15.5)/4095;
    printf("VIN %.2f\r\n",Vin);
    HAL_Delay(1000);
}
```

7.1.2 PENGATURAN PWM DENGAN ADC

Dalam sistem eletronika, banyak alat yang menggunakan komponen analog untuk melakukan pengaturan, misal potensiometer analog digunakan untuk mengatur volume atau nada bass di perangkat audio. Walaupu Xn mungkinK potensiometer tersebut sudah digantikan oleh *rotary encoder*. Agar potensiometer bisa digunakan dalam sistem berbasis

mikrokontroler, nilai potensiometer tersebut harus dikonversi ke bentuk digital, digunakanlah ADC. Tentu saja, ADC tidak akan bisa membaca nilai resistansi melainkan nilai tegangan yang dihubungkan ke potensiometer tersebut. Jika potentiometer tersebut diputar, tegangan di kaki tengah potensiometer tersebut akan berubah sesuai posisi/putaran potensiometer tersebut. Tegangan inilah yang dibaca oleh ADC yang ada di mikrokontroler.



Gambar 7.3 Rangkaian Antarmuka Potensiometer

Contoh program berikut merupakan contoh program gabungan antara pengendalian LED dengan sinyal PWM dengan pembacaan potensiometer. Sehingga potensiometer bisa digunakan untuk mengatur kecerahan dari LED dengan cara mengatur lebar pulsa sinyal PWM. Potensiometer dibaca dengan ADC1_IN10 (pin PC0) sedangkan LED terhubung ke pin PA0 yang difungsikan sebagai keluaran PWM TIM2_CH1. Penjelasan mengenai program PWM bisa dilihat di sub-bab 6.3.2.

Program utama dari contoh program ini, adalah membaca nilai ADC dan mengubahnya menjadi nilai siklus kerja dalam persentase (variabel dutycycle). Hasil pembacaan ini kemudian digunakan untuk mengatur sinyal PWM yang terhubung ke LED. Sehingga kecerahan LED bisa diatur dengan potensiometer. Untuk mengubah nilai ADC menjadi nilai siklus kerja, digunakan rumus:

$$\text{dutycycle} = (\text{uhADCxConvertedValue} * 100) / 4095$$

uhADCxConvertedValue adalah nilai yang dibaca dari ADC, 4095 merupakan nilai maksimal pembaca ADC (12 bit).

Listing Program 7.3 Program Utama

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
```

```
/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */
if (HAL_ADC_Start(&hadc1)!=HAL_OK)
{
    Error_Handler();
}

//Poll ADC, wait until conversion finish
HAL_ADC_PollForConversion(&hadc1, 10);
if((HAL_ADC_GetState(&hadc1) & HAL_ADC_STATE_EOC_REG) ==
HAL_ADC_STATE_EOC_REG)
{
    //Read ADC Value
    uhADCxConvertedValue = HAL_ADC_GetValue(&hadc1);
}

//Convert to dutycycle
dutycycle = (uhADCxConvertedValue *100)/4095;
printf("Dutycycle %d\r\n",dutycycle);
sConfigOC.Pulse = (2499 * dutycycle)/100;
if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC,
TIM_CHANNEL_1) != HAL_OK)
{
    Error_Handler();
}

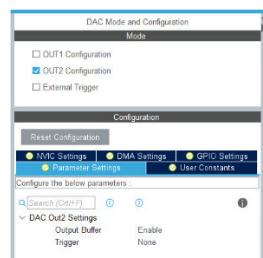
if (HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1) != HAL_OK)
{
    /* PWM Generation Error */
    Error_Handler();
}
HAL_Delay(100);
```

7.2 BERMAIN-MAIN DENGAN DAC

DAC bekerja berkebalikan dengan ADC, DAC akan mengubah nilai digital menjadi nilai analog (dalam hal ini tegangan) yang sesuai, dengan tegangan DAC 3.3V maka DAC STM32F207 bisa diprogram untuk mengeluarkan tegangan analog dari 0 sampai 3.3V dalam resolusi 12 bit. DAC juga bisa diprogram secara langsung untuk menghasilkan sinyal noise dan gelombang segitiga, pustaka HAL sudah menyediakan fungsi untuk itu.

7.2.1 PEMBANGKITAN TEGANGAN DC DENGAN DAC

Seperti telah dijelaskan, DAC bekerja dengan mengubah nilai digital menjadi nilai analog (tegangan). Contoh program berikut akan menunjukkan bagaimana DAC yang dimiliki oleh STM32F207 menghasilkan tegangan DC dari 0 - 3,3V. Tegangan yang dihasilkan diatur oleh 2 buah saklar yang akan menaikan atau menurunkan tegangan yang dihasilkan.



Gambar 7.4 Pengaturan DAC

DAC menggunakan DAC channel 2 (pin PA5) dan 2 buah saklar terhubung ke pin PG0 dan PG1. Oleh karena dicontoh program ini hanya akan menghasilkan sinyal DC, maka pemicu (trigger) tidak digunakan.

Listing Program 7.4 Inisialisasi DAC

```
void MX_DAC_Init(void)
{
    DAC_ChannelConfTypeDef sConfig = {0};

    /** DAC Initialization
    */
    hdac.Instance = DAC;
    if (HAL_DAC_Init(&hdac) != HAL_OK)
    {
        Error_Handler();
    }
    /** DAC channel OUT2 config
    */
    sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
    sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
    if (HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_2) != HAL_OK)
    {
        Error_Handler();
    }
}
```

Setelah diinisialisasi, DAC belum mengeluarkan tegangan karena proses konversi belum dimulai dan register datanya belum diisi. Dalam contoh program ini, DAC akan diprogram untuk bekerja di 8 bit (0 – 255). Sesaat setelah proses inisialisasi, data register dengan nilai 127 sehingga ketika konversi dimulai, pin DAC akan mengeluarkan tegangan DC sekitar 1,6V.

Listing Program 7.5 Pemberian Nilai Awal DAC

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DAC_Init();
/* USER CODE BEGIN 2 */
DAC_Value = 127;
if(HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_8B_R,
DAC_Value) != HAL_OK)
{
    /* Setting value Error */
    Error_Handler();
}

/*##-4- Enable DAC Channel1
#####
if(HAL_DAC_Start(&hdac, DAC_CHANNEL_2) != HAL_OK)
{
    /* Start Error */
    Error_Handler();
}

/* USER CODE END 2 */
```

Di loop utama, program akan membaca 2 buah saklar, jika KEY1 ditekan, maka nilai DAC akan dinaikkan 20, sedangkan jika KEY2 yang ditekan, nilai DAC akan dikurangi 20. Jika salah satu saklar ditekan, flag *KeyPressed* akan bernilai 1. Proses pembacaan saklar, tentu saja, menggunakan anti bouching dengan delay.

Listing Program 7.6 Pembacaan Saklar

```
//Read Up Key
if(!HAL_GPIO_ReadPin(KEY1_GPIO_Port, KEY1_Pin)){
    keypresstimer=0;
    while(!HAL_GPIO_ReadPin(KEY1_GPIO_Port, KEY1_Pin)){
        keypresstimer++;
        HAL_Delay(1);
    }
    if(keypresstimer>10){
        KeyPressed=1;
        DAC_Value+=20;
    }
}
```

```

    }

    if(!HAL_GPIO_ReadPin(KEY2_GPIO_Port, KEY2_Pin)){
        keypresstimer=0;
        while(!HAL_GPIO_ReadPin(KEY2_GPIO_Port, KEY2_Pin)){
            keypresstimer++;
            HAL_Delay(1);
        }
        if(keypresstimer>10){
            KeyPressed=1;
            DAC_Value-=20;
        }
    }
}

```

Ketika ada saklar yang ditekan, register data DAC akan diisi dengan variabel *DAC_Value*, dan kemudian proses konversi akan dimulai lagi. Oleh karena DAC bekerja di 8 bit, maka tegangan DC yang dihasilkan adalah:

$$\text{Tegangan DAC} = \frac{\text{DAC_Value}}{255} \times 3.3$$

Jika diinginkan DAC bekerja di resolusi 12 bit, tinggal mengganti parameter *DAC_ALIGN_8B_R* dengan *DAC_ALIGN_12B_R* saat memanggil fungsi *HAL_DAC_SetValue*. Dan pembagi di rumus tegangan DAC akan menjadi 4095.

Listing Program 7.7 Proses Konversi DAC

```

if(KeyPressed){
    if(HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_8B_R,
    DAC_Value) != HAL_OK)
    {
        /* Setting value Error */
        Error_Handler();
    }

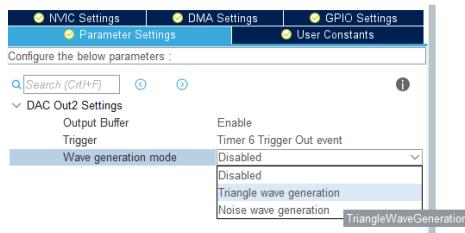
    if(HAL_DAC_Start(&hdac, DAC_CHANNEL_2) != HAL_OK)
    {
        /* Start Error */
        Error_Handler();
    }

    KeyPressed=0;
}

```

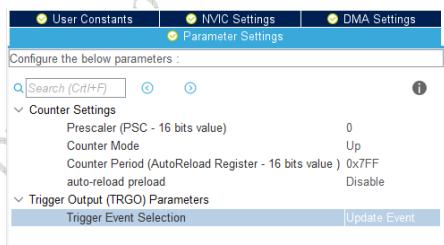
7.2.2 PEMBANGKITAN SINYAL SEGITIGA DAN NOISE

DAC STM32F207 dilengkapi dengan register geser LFSR dan pembangkit gelombang segitiga (TRIANGLEx). LFSR bisa digunakan untuk membangkitkan sinyal noise sedangkan TRIANGLEx, sesuai dengan namanya, bisa digunakan untuk membangkitkan gelombang segitiga. Untuk bekerja di mode ini, DAC membutuhkan sinyal pemicu yang bisa berasal dari timer (sinyal TRGO) atau pun software.



Gambar 7.5 Pengaturan DAC untuk Pembangkit Gelombang

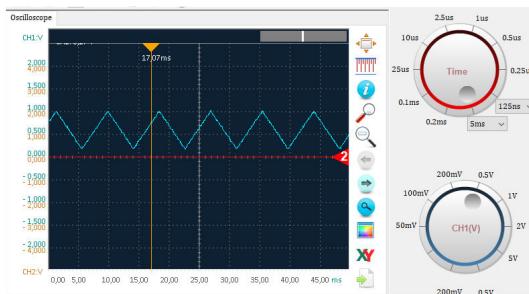
Tidak semua timer yang dimiliki oleh STM32F207 bisa dijadikan pemicu DAC, hanya TIM2, TIM4, TIM5, TIM6, TIM7 dan TIM8 yang bisa dipakai. Timer yang digunakan sebagai pemicu DAC, akan bekerja sebagai basis pewaktuan dengan keluaran pemicunya diaktifkan, dan akan aktif setiap ada update (*update event*). Nilai counter periode dari timer akan menentukan frekuensi, terutama sinyal segitiga, sinyal yang dihasilkan. Satu buah saklar yang terhubung ke pin PG1 digunakan untuk memilih antara sinyal noise atau segitiga dan bisa diamati melalui osiloskop di pin keluaran DAC channel 2 (PA5).



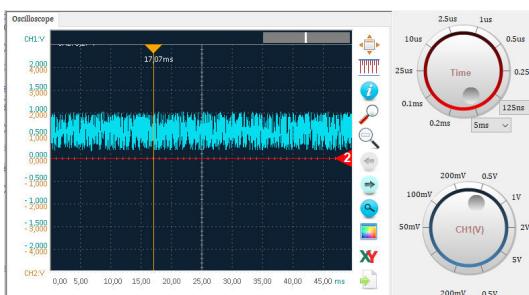
Gambar 7.6 Pengaturan Timer untuk Sinyal Pemicu DAC

DAC diinisialisasi awal sebagai pembangkit gelombang segitiga. Fungsi inisialisasi DAC yang dihasilkan STM32CubeMX (MX_DAC_Init) belum mengaktifkan DAC begitu juga TIM6 yang digunakan, sehingga harus

diaktifkan di program utama. Variabel *TriangleWave* akan mengatur bentuk gelombang yang dihasilkan, dan diubah melalui penekanan saklar. Jika bernilai 1, maka DAC akan diprogram untuk membangkitkan sinyal segitiga, dan jika bernilai 0 akan membangkitkan sinyal noise.



Gambar 7.7 Gelombang Segitiga yang dihasilkan



Gambar 7.8 Sinyal Noise yang Dihasilkan

Listing Program 7.8 Fungsi Utama

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DAC_Init();
MX_TIM6_Init();
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start(&htim6);
DAC_Start();
TriangleWave=1;
/* USER CODE END 2 */
```

HAL sudah menyediakan fungsi untuk mengaktifkan DAC sebagai pembangkit noise atau pembangkit sinyal segitiga:

Listing Program 7.9 Fungsi Pembangkit Sinyal Segitiga

```
if (HAL_DACEx_TriangleWaveGenerate(&hdac, DAC_CHANNEL_2,
DAC_TRIANGLEAMPLITUDE_1023) != HAL_OK)
{
    Error_Handler();
```

```
}
```

Listing Program 7.10 Fungsi Pembangkit Sinyal Noise

```
if(HAL_DACEx_NoiseWaveGenerate(&hdac, DAC_CHANNEL_2,  
DAC_LFSRUNMASK_BITS9_0) != HAL_OK)  
{  
    Error_Handler();  
}
```

Kedua fungsi ini membutuhkan parameter pointer DAC, channel yang digunakan dan amplitudo sinyal. Sedangkan frekuensi sinyal ditentukan oleh timer yang digunakan sebagai pemicu. Amplituda dan frekuensi di contoh program ini tidak bisa diatur.

by Kang U2Man (u_2man@yahoo.co.id)

Bab 8

BERMAIN-MAIN DENGAN DMA

DMA (*Direct Memory Access*) merupakan sebuah mekanisme yang dimiliki oleh prosesor/mikrokontroler untuk melakukan transfer data dari memori ke memori, memori ke periperal dan sebaliknya tanpa melibatkan CPU. Di dalam program, ketika akan melakukan transfer data, misal antar periperal dengan memori, program tinggal mengaktifkan kendali DMA untuk periperal tersebut, mengaktifkan interupsi dan memulai transfer melalui DMA. CPU kemudian bisa diprogram untuk menjalankan fungsi lain, tidak perlu menunggu sampai transfer data selesai. Tetapi, ketika akan melakukan transfer data kembali, CPU harus mengecek apakah proses transfer data sebelumnya sudah selesai atau belum.

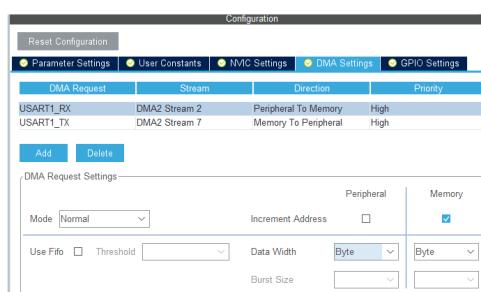
Hampir semua periperal yang dimiliki oleh STM32 memiliki atau terhubung dengan DMA. Periperal yang membutuhkan kecepatan tinggi seperti USB, ethernet atau pun SDIO dipastikan mempunyai DMA. Di bab ini akan ditunjukkan bagaimana menggunakan DMA untuk menerima atau mengirim data melalui UART dan I2C dan juga penggunaan DMA dengan DAC untuk membangkitkan sinyal sinus, gigi gergaji dan lain-lain. Di bab-bab selanjutnya mungkin juga akan diberikan contohnya, seperti ketika menggunakan SDIO (mikro SD) dengan FreeRTOS, STM32CubeMX akan secara otomatis mengaktifkan DMA untuk SDIO. Karena memang DMA akan lebih efektif jika program menggunakan RTOS (*Real Time Operation System*).

8.1 UART DAN I2C DENGAN DMA

Contoh program berikut hampir sama dengan pengendalian mode kerja LED melalui UART, mode kerja kemudian akan disimpan di EEPROM I2C. Bedanya, proses pengiriman dan penerimaan data dari UART dilakukan dengan DMA, begitu juga ketika menyimpan atau membaca mode kerja LED dari EEPROM akan melibatkan DMA.

8.1.1 TRANSFER DATA UART DENGAN DMA

Setiap periperal telah dipetakan ke kendali DMA, kanal dan *stream* dari kendali DMA yang dipakai. Sebagai contoh USART1 dipetakan ke kendali DMA2 kanal 4, untuk RX menggunakan stream2 atau stream5 sedangkan TX menggunakan stream7. Arah transfer data, tentu saja, akan berasal dari periperal ke memori untuk RX dan dari memori ke periperal untuk TX. DMA TX dan RX bekerja secara independen, artinya DMA bisa diaktifkan keduanya (RX dan TX) atau hanya salah satu yang diaktifkan.

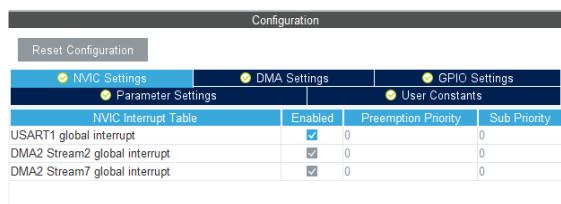


Gambar 8.1 Pengaturan DMA untuk USART1

DMA sendiri bisa diatur untuk mode buffer normal atau *circular*. Mode *circular* lebih cocok untuk digunakan oleh data kontinyu seperti ADC, sedangkan untuk USART mode normal lebih tepat. Data yang dikirim USART biasanya dalam bentuk ASCII atau 8 bit, sehingga ukuran data yang digunakan adalah byte. USART STM32F2 tidak mempunyai buffer khusus, register datanya hanya 1 byte, sehingga penambahan alamat terjadi di memori bukan di periperal.

Seperti dijelaskan di awal bab ini, DMA bekerja bersamaan dengan interupsi, ketika ingin menggunakan DMA, interupsi periperal yang bersangkutan juga harus diaktifkan. STM32CubeMX akan mengaktifkan

interupsi DMA secara otomatis ketika DMA tersebut diaktifkan. Sedangkan untuk periperal, interupsinya harus diaktifkan secara manual.



Gambar 8.2 Pengaturan Interupsi

Program pengendalian LED pada dasarnya akan menerima data dari UART, men-set mode LED, menyimpan ke EEPROM dan membalas melalui UART bahwa perintah telah diterima dengan menggunakan printf. Dalam program tersebut, printf diarahkan fungsi __io_putchar untuk mengirim 1 byte ke UART. Mengirim 1 byte data menggunakan DMA tentu tidak akan mendapatkan keuntungan dari DMA, dengan kata lain tidak efisien. Secara teori, semakin besar data yang dikirim/diterima, semakin efisien penggunaan DMA tersebut.

Seperti telah dijelaskan bahwa fungsi printf akan memanggil fungsi write yang ada di file syscall.c, kemudian fungsi write akan memanggil fungsi HAL untuk mengirim data melalui UART. Untuk pengiriman dengan DMA, fungsi write tinggal memanggil pengiriman UART dengan DMA (HAL_UART_Transmit_DMA). Pengiriman data dengan DMA tidak akan melibatkan CPU begitu transfer data dimulai, namun sebelum mengirim data ke kendali DMA perlu dicek dahulu apakah pada saat itu DMA sedang melakukan transfer data atau tidak.

Sebuah variabel, *UARTSendFinish* digunakan untuk mengecek status pengiriman DMA, jika variabel ini bernilai 1, maka kendali DMA sedang tidak melakukan transfer data, jika bernilai 0, maka program harus menunggu sampai *UARTSendFinish* bernilai 1 atau jika terjadi *timeout* (untuk menghindari program *hang*). Variabel *UARTSendFinish* diberi nilai 0 sesaat sebelum memanggil fungsi pengiriman data melalui DMA, dan diberi nilai 1 ketika reset, karena memang setelah reset belum terjadi pengiriman melalui DMA.

Listing Program 8.1 Fungsi Write dengan DMA

```
int _write(int file, char *ptr, int len)
{
    uint32_t WaitTimeOut=0;
```

```

        while(!UARTSendFinish){
            if(WaitTimeOut++>100)
                return len;
            HAL_Delay(1);
        }

        UARTSendFinish=0;
        if(HAL_UART_Transmit_DMA(&huart1, (uint8_t*)ptr, len)!= HAL_OK)
        {
            /* Transfer error in transmission process */
            Error_Handler();
        }

        return len;
    }
}

```

Fungsi `HAL_UART_Transmit_DMA` akan memanggil fungsi untuk mengaktifkan DMA (`HAL_DMA_Start_IT`), fungsi yang akan dipanggil ketika melakukan transfer data melalui DMA, baik antara periperal dan memori, atau antara memori dan memori. Fungsi ini membutuhkan parameter berupa pointer kepada channel DMA yang digunakan, dalam hal ini DAM2 Stream2 dan channel 4, alamat asal dan tujuan, yaitu register data UART (`huart->Instance->DR`) dan pointer data (variabel `tmp`). Setelah fungsi ini, transfer data DMA dimulai sesuai dengan paramter yang dimasukan.

Listing Program 8.2 Fungsi Pengiriman UART dengan DMA

```

HAL_StatusTypeDef HAL_UART_Transmit_DMA(UART_HandleTypeDef *huart,
                                         uint8_t *pData, uint16_t Size)
{
    uint32_t *tmp;

    /* Check that a Tx process is not already ongoing */
    if (huart->gState == HAL_UART_STATE_READY)
    {
        if ((pData == NULL) || (Size == 0U))
        {
            return HAL_ERROR;
        }

        /* Process Locked */
        __HAL_LOCK(huart);

        huart->pTxBuffPtr = pData;
        huart->TxXferSize = Size;
        huart->TxXferCount = Size;

        huart->ErrorCode = HAL_UART_ERROR_NONE;
        huart->gState = HAL_UART_STATE_BUSY_TX;
    }
}

```

```
/* Set the UART DMA transfer complete callback */
huart->hdmatx->XferCpltCallback = UART_DMATransmitCplt;

/* Set the UART DMA Half transfer complete callback */
huart->hdmatx->XferHalfCpltCallback = UART_DMATxHalfCplt;

/* Set the DMA error callback */
huart->hdmatx->XferErrorCallback = UART_DMAError;

/* Set the DMA abort callback */
huart->hdmatx->XferAbortCallback = NULL;

/* Enable the UART transmit DMA stream */
tmp = (uint32_t *)&pData;
HAL_DMA_Start_IT(huart->hdmatx, *(uint32_t *)tmp,
(uint32_t)&huart->Instance->DR, Size);

/* Clear the TC flag in the SR register by writing 0 to it */
__HAL_UART_CLEAR_FLAG(huart, UART_FLAG_TC);

/* Process Unlocked */
__HAL_UNLOCK(huart);

/* Enable the DMA transfer for transmit request by setting the
DMAT bit
in the UART CR3 register */
SET_BIT(huart->Instance->CR3, USART_CR3_DMAT);

return HAL_OK;
}
else
{
    return HAL_BUSY;
}
}
```

Sesuai dengan namanya, fungsi `HAL_DMA_Start_IT` akan mengaktifkan interupsi DMA yaitu interupsi saat transfer data selesai (*transfer complete*), ketika ada error (*transfer error*) dan ketika ketika *mode direct error* dan juga interupsi ketika data transfer setelah selesai setengahnya (*half complete*). Interupsi ini akan dilayani melalui layanan interupsi yang telah ditentukan. Ketika terjadi interupsi transfer data selesai, layanan interupsi akan memanggil fungsi callback yang telah didefinisikan di fungsi `HAL_UART_Transmit_DMA` yaitu fungsi `UART_DMATransmitCplt`. Fungsi ini kemudian akan mengaktifkan interupsi *UART transmit complete* (TC).

Listing Program 8.3 Fungsi UART_DMATransmitCplt

```

static void UART_DMATransmitCplt(DMA_HandleTypeDef *hdma)
{
    UART_HandleTypeDef *huart = (UART_HandleTypeDef)
*)((DMA_HandleTypeDef *)hdma)->Parent;
    /* DMA Normal mode*/
    if ((hdma->Instance->CR & DMA_SxCR_CIRC) == 0U)
    {
        huart->TxXferCount = 0x00U;

        /* Disable the DMA transfer for transmit request by setting the
        DMAT bit
        in the UART CR3 register */
        CLEAR_BIT(huart->Instance->CR3, USART_CR3_DMAT);

        /* Enable the UART Transmit Complete Interrupt */
        SET_BIT(huart->Instance->CR1, USART_CR1_TCIE);

    }
    /* DMA Circular mode */
    else
    {
#if (USE_HAL_UART_REGISTER_CALLBACKS == 1)
        /* Call registered Tx complete callback */
        huart->TxCpltCallback(huart);
#else
        /*Call legacy weak Tx complete callback*/
        HAL_UART_TxCpltCallback(huart);
#endif /* USE_HAL_UART_REGISTER_CALLBACKS */
    }
}

```

Alasan kenapa saat interupsi DMA komplit terjadi program malah mengaktifkan interupsi UART transfer komplit adalah karena pada saat terjadi interupsi DMA komplit tugas kendali DMA mentransfer data dari memori ke periperal (UART) sudah selesai, tetapi proses transfer data belum benar-benar selesai karena masih menunggu proses transfer di UART. Oleh karena itu, untuk mengetahui proses transfer data benar-benar selesai adalah dengan mengaktifkan interupsi UART TC.

Ketika proses pengiriman data di UART selesai, interupsi dibangkitkan. Layanan interupsi kemudian akan memanggil fungsi callback saat proses transfer selesai, yaitu fungsi *HAL_UART_TxCpltCallback*. Di fungsi inilah variabel *UARTSendFinish* di-update menjadi 1, yang menandakan bahwa proses pengiriman melalui DMA telah selesai. Kendali DMA telah siap untuk menerima data selanjutnya. Jadi dalam proses pengiriman data

melalui UART dan DMA akan melibatkan interupsi DMA dan interupsi UART itu sendiri.

Listing Program 8.4 Fungsi HAL_UART_TxCpltCallback

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart){  
    UARTSendFinish=1;  
}
```

Pada proses penerimaan, fungsi penerimaan data UART melalui DMA (*HAL_UART_Receive_DMA*) harus segera dipanggil setelah UART dan DMA diinisialisasi, karena data bisa datang setiap saat. Setelah data diterima, maka fungsi *HAL_UART_Receive_DMA* harus dipanggil kembali agar bisa menerima data berikutnya.

Seperti dijelaskan di program sebelumnya, panjang data untuk pengaturan mode kerja LED tidak tetap. Sedangkan fungsi penerimaan data UART melalui DMA membutuhkan panjang data tersebut, artinya interupsi DMA baru terjadi ketika panjang data sudah sesuai dengan panjang data yang ditetapkan saat memanggil fungsi penerimaan DMA. Oleh karena itu, diprogram ini, seperti halnya saat contoh program menggunakan metode interupsi, panjang data penerimaan DMA ditentukan 1 byte. Ketika ada data 1 byte diterima, DMA akan membangkitkan interupsi dan layanan interupsinya akan memanggil fungsi *HAL_UART_RxCpltCallback*. Berbeda dengan pengiriman data UART dengan DMA, fungsi ini tidak dipanggil oleh interupsi UART, tetapi benar-benar oleh interupsi DMA. Karena memang tidak ada pengaktifan interupsi penerimaan melalui UART (RXNE).

Listing Program 8.5 Fungsi Callback Penerimaan Data

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){  
  
    //Save received data to buffer  
    if((UartRxBuffer!=0x0D)&&(UartRxBuffer!=0x0A)){  
        pcbuffer[pcbuffercounter++]=UartRxBuffer;  
    }  
  
    if(UartRxBuffer==0x0A){  
        pcframe=1;  
    }  
  
    //Start for next data
```

```

if(HAL_UART_Receive_DMA(huart, (uint8_t*)&UartRxBuffer,
1)!=HAL_OK){
    Error_Handler();
}
}

```

Ketika ada data, data yang diterima disimpan di sebuah buffer array (*pcbuffer*) jika diterima byte 0x0A *pcframe* akan di-set, sehingga program utama akan memanggil fungsi *pc_link* untuk memproses perintah yang diterima. Fungsi *HAL_UART_Receive_DMA* dipanggil kembali setiap menerima data agar bisa menerima data selanjutnya.

8.1.2 TRANSFER DATA I2C DENGAN DMA

I2C2, yang terhubung dengan EEPROM, dipetakan ke DMA1 kanal 7. Untuk pembacaan menggunakan stream 2 dan untuk penulisan menggunakan stream 7. Kendali DMA diatur dengan lebar data byte dengan mode kerja normal. Arah data transfer sendiri diatur seperti halnya UART, dari periperal ke memori untuk penerimaan, dan dari memori ke periperal untuk penerimaan. I2C2 bekerja dengan interupsi *event* dan error yang diaktifkan.



Gambar 8.3 Pengaturan DMA untuk I2C2

Untuk penulisan dan pembacaan EEPROM dengan DMA, pustaka HAL sudah menyediakan fungsinya. Fungsi tersebut hampir sama dengan fungsi tanpa DMA, bedanya fungsi dengan DMA tidak membutuhkan *timeout* di parameteranya.

Listing Program 8.6 Penulisan dan Pembacaan EEPROM dengan DMA

```

uint8_t EEPROM_Read(uint16_t Addr,uint8_t *buf,uint16_t num){
    uint8_t temp;
    uint32_t WaitTimeout=0;

    temp = HAL_I2C_Mem_Read_DMA(&hi2c2, EEPROM_ADDR, Addr,
I2C_MEMADD_SIZE_16BIT, buf, num);
    while (HAL_I2C_GetState(&hi2c2) != HAL_I2C_STATE_READY){
        if(WaitTimeout>100){
            break;
        }
    }
}

```

```
        }
        HAL_Delay(1);
    }
    return temp;
}

uint8_t EEPROM_Write(uint16_t Addr,uint8_t *buf,uint16_t num){
    uint8_t temp;
    uint32_t WaitTimeout=0;

    temp = HAL_I2C_Mem_Write_DMA(&hi2c2, EEPROM_ADDR, Addr,
I2C_MEMADD_SIZE_16BIT, buf, num);
    while (HAL_I2C_GetState(&hi2c2) != HAL_I2C_STATE_READY){
        if(WaitTimeout++>100)
            break;
        HAL_Delay(1);
    }
    return temp;
}
```

Fungsi untuk penulisan dan pembacaan EEPROM dengan DMA ini tidak menggunakan variabel status seperti halnya variabel *UARTSendFinish*. Sehingga untuk I2C tidak digunakan fungsi callback. Sebagai gantinya, setelah membaca atau menulis data ke EEPROM, program menunggu dulu sampai proses transfer data selesai, dalam hal ini menunggu status I2C2 menjadi siap.

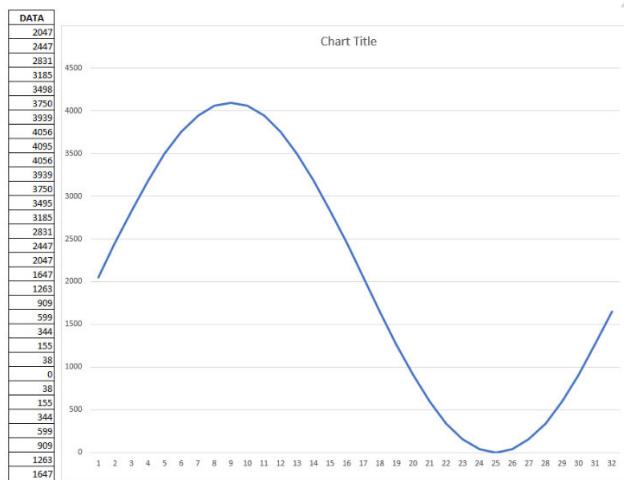
Dari contoh I2C ini, terlihat sepertinya DMA tidak berguna karena program tetap harus menunggu sampai proses transfer data selesai. Sebenarnya untuk I2C juga bisa dibuat seperti halnya untuk transfer data dengan UART, hanya saja di contoh I2C ini untuk menunjukkan bahwa setelah transfer data dimulai, program bisa menunggu sampai proses selesai, atau mengerjakan fungsi lain. Apabila program telah menggunakan RTOS, maka ketika sedang menunggu dan program memanggil fungsi delay, maka secara otomatis program akan *switch* untuk mengerjakan task yang lain, dengan catatan fungsi delay yang dipanggil merupakan fungsi delay RTOS.

8.2 GENERATOR SINYAL DENGAN DAC

Di bab sebelumnya sudah diberikan contoh bagaimana menggunakan DAC untuk membangkitkan sinyal noise dan sinyal segitiga. Kedua sinyal ini memang didukung secara hardware oleh DAC, artinya di

dalam DAC sendiri sudah ada fitur untuk membangkitkan kedua sinyal tersebut. Di bab ini akan ditunjukkan bagaimana menggunakan DAC untuk membangkitkan sinyal seperti sinyal sinus, gigi gergaji dan lain-lain. Dan juga bagaimana mengatur frekuensi sinyal yang dibangkitkan dengan mengatur periodik timer yang digunakan sebagai trigger DAC. Tentu saja, sesuai tema bab ini, DAC akan diprogram untuk bekerja dengan DMA.

Untuk membangkitkan bentuk sinyal yang diinginkan, maka DAC perlu diberi data yang ketika diterjemahkan oleh DAC akan membentuk tegangan atau sinyal yang diinginkan. Gambar di bawah ini menunjukkan bagaimana grafik gelombang sinus dibentuk dari sebuah tabel data. Kalau data-data dari tabel dikirimkan ke DAC, maka DAC juga akan mengeluarkan tegangan yang berbentuk sinus. Kecepatan pengiriman data akan menentukan frekuensi gelombang sinus yang diinginkan.



Gambar 8.4 Grafik Gelombang Sinus

Dalam contoh program berikut ini, DAC akan diprogram untuk menghasilkan gelombang sinus, gigi gergaji (*saw tooth*), segitiga, kotak dan gelombang eskalator. Masing-masing gelombang memerlukan tabel data, sehingga diperlukan 5 tabel data. Tabel data ini dibentuk dengan data 16 bit (*half word*), namun oleh karena DAC STM32F207 adalah 12 bit, maka nilai maksimal untuk tabel data adalah $2^2 \cdot 1$ (4095). Data-data inilah yang akan dikirimkan ke DAC. Untuk data gelombang kotak, hanya terdiri atas 2 data, karena memang gelombang kotak hanya memerlukan data 0 (0V) dan 4095 (3,3V) seperti meng-*toggle* GPIO. DAC tidak bisa

menghasilkan tegangan negatif, oleh karena itu keluarannya mempunyai tegangan offset (sebesar $\frac{1}{2}$ tegangan referensi DAC). Untuk menghilangkannya bisa digunakan sebuah kapasitor kopling.

Listing Program 8.7 Tabel Data Sinyal

```
uint16_t Sine_Wave_Data_Table[32] ={
    2047, 2447, 2831, 3185, 3498, 3750, 3939, 4056,
    4095, 4056, 3939, 3750, 3495, 3185, 2831, 2447,
    2047, 1647, 1263, 909, 599, 344, 155, 38,
    0, 38, 155, 344, 599, 909, 1263, 1647
};

uint16_t Sawtooth_Wave_Data_Table[32]={
    0, 132, 264, 396, 528, 660, 792, 924,
    1057, 1189, 1321, 1453, 1585, 1717, 1849, 1981,
    2113, 2245, 2377, 2509, 2641, 2773, 2905, 3037,
    3170, 3302, 3434, 3566, 3698, 3830, 3962, 4095
};

uint16_t Triangle_Wave_Data_Table[32]={
    0, 256, 512, 768, 1024, 1279, 1535, 1791,
    2047, 2303, 2559, 2815, 3071, 3326, 3582, 3838,
    4095, 3838, 3582, 3326, 3071, 2815, 2559, 2303,
    2047, 1791, 1535, 1279, 1024, 768, 512, 256
};

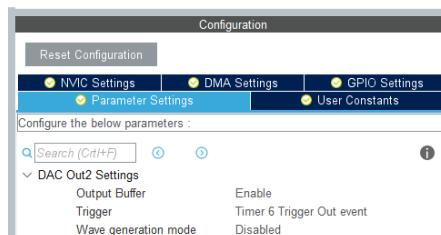
uint16_t Square_Wave_Data_Table[2]={
    0, 4095
};

uint8_t Escalator_Wave_Data_Table[6]={
    0x0, 0x33, 0x66, 0x99, 0xCC, 0xFF
};
```

DMA akan digunakan untuk mengirimkan data-data tersebut ke DAC, sehingga program tidak perlu melakukan *looping* untuk mengirimkan data-data tersebut ke DAC. Dalam penggunaan DMA, ada 2 mode yang bisa digunakan, yaitu mode normal dan mode circular. Pada mode normal, kendali DMA hanya mentransfer data sekali, artinya kalau buffer sudah habis dikirim, DMA harus diperintah kembali untuk melakukan transfer berikutnya. Sedangkan pada mode circular, DMA akan terus menerus mentransfer data sampai diberi perintah untuk berhenti. Untuk membangkitkan gelombang ini, sepertinya lebih cocok menggunakan mode circular.

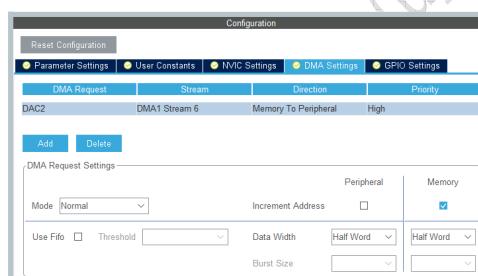
8.2.1 DMA MODE NORMAL

DAC bekerja dengan pemicu dari Timer 6, dengan fungsi pembangkitan gelombang tidak aktif. Sementara Timer 6 bekerja seperti contoh program sebelumnya (pembangkitan sinyal noise dan gelombang segitiga). Interupsi keduanya diaktifkan. Dalam contoh program ini digunakan 3 buah saklar, 1 digunakan untuk mengubah bentuk gelombang sedangkan kedua saklar yang lain digunakan untuk mengubah frekuensi sinyal yang dihasilkan.



Gambar 8.5 Pengaturan DAC

Pada contoh yang pertama ini DMA bekerja pada mode normal. Data yang dikirim mempunyai lebar 16 bit (*half word*). Interupsi DMA secara default akan diaktifkan.



Gambar 8.6 Pengaturan DMA DAC Mode Normal

Untuk mengirimkan data ke DAC menggunakan DMA, pustaka HAL sudah menyediakan fungsi *HAL_DAC_Start_DMA*. Fungsi ini memerlukan parameter pointer ke DAC yang digunakan, kanalnya, pointer data, panjang data dan pengaturan data. Dalam program ini pointer data menunjuk ke tabel yang sedang sesuai dengan bentuk gelombang yang akan dibangkitkan. Variabel *Waveform* menentukan tabel mana yang akan dikirim. Nilai *Waveform* ditentukan sebagai berikut:

1. Bernilai 0, jika akan dibangkitkan gelombang sinus.

2. Bernilai 1, untuk gelombang gigi gergaji.
3. Bernilai 2, untuk gelombang eskalator.
4. Bernilai 3, untuk gelombang segitiga.
5. Bernilai 4, untuk gelombang kotak

Listing Program 8.8 Pembangkit Gelombang

```
void DAC_Signal_Generator(void){  
    switch (WaveForm){  
        case 0:  
            //Sine Wave  
            if(HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_2,  
(uint32_t*)Sine_Wave_Data_Table, 32, DAC_ALIGN_12B_R) != HAL_OK)  
            {  
                /* Start DMA Error */  
                Error_Handler();  
            }  
  
            break;  
        case 1:  
            //Sawtooth  
            if(HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_2,  
(uint32_t*)Sawtooth_Wave_Data_Table, 32, DAC_ALIGN_12B_R) != HAL_OK)  
            {  
                /* Start DMA Error */  
                Error_Handler();  
            }  
  
            break;  
        case 2:  
            //Escalator  
            if(HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_2,  
(uint32_t*)Escalator_Wave_Data_Table, 6, DAC_ALIGN_8B_R) != HAL_OK)  
            {  
                /* Start DMA Error */  
                Error_Handler();  
            }  
  
            break;  
        case 3:  
            //Trianglewave:  
            if(HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_2,  
(uint32_t*)Triangle_Wave_Data_Table, 32, DAC_ALIGN_12B_R) != HAL_OK)  
            {  
                /* Start DMA Error */  
                Error_Handler();  
            }  
  
            break;  
        case 4:
```

```

        //Square wave
        if(HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_2,
(uint32_t*)Square_Wave_Data_Table, 2, DAC_ALIGN_12B_R) != HAL_OK)
{
    /* Start DMA Error */
    Error_Handler();
}

break;
}
}

```

Nilai *Waveform* ini akan diperbarui oleh sebuah saklar, sehingga bentuk gelombang bisa diubah-ubah melalui saklar ini.

Listing Program 8.9 Mengubah Bentuk Gelombang

```

if(!HAL_GPIO_ReadPin(KEY1_GPIO_Port, KEY1_Pin)){
    keypresstimer=0;
    while(!HAL_GPIO_ReadPin(KEY1_GPIO_Port, KEY1_Pin)){
        keypresstimer++;
        HAL_Delay(1);
    }
    if(keypresstimer>10){
        WaveForm++;
        if(WaveForm>4)
            WaveForm=0;
    }
}

```

Ketika kendali DMA sudah selesai mengirimkan data, kendali DMA akan membangkitkan interupsi transfer komplit. Kemudian berhenti, karena kendali DMA bekerja di mode normal. Agar gelombang yang dihasilkan kontinyu, data perlu dikirim kembali. Tabel data hanya cukup untuk menghasilkan 1 gelombang. Interupsi transfer komplit digunakan untuk mengirimkan kembali data tabel gelombang, dengan memanggil kembali fungsi *DAC_Signal_Generator*. Kondisi ini mirip dengan kendali DMA untuk penerimaan UART, yang harus dipanggil kembali setiap kali menerima data.

Listing Program 8.10 Fungsi Callback Interupsi DMA

```

void HAL_DACEx_ConvCpltCallbackCh2(DAC_HandleTypeDef *hdac){
    DAC_Signal_Generator();
}

```

Frekuensi gelombang bisa diatur dengan mengatur periode timer yang digunakan untuk memicu DAC. Timer bekerja sebagai basis pewaktuan, dengan event update diaktifkan. Setiap kali timer mengalami overflow

(interupsi update aktif), kendali DMA akan mengirim 1 data yang ada di tabel. Jika tabel, misal untuk gelombang sinus, mempunyai ukuran $32 \times$ half word, maka untuk 1 gelombang sinus dibutuhkan waktu $32 \times$ interupsi timer. Jika timer bekerja di 1 mili detik, maka gelombang sinus akan mempunyai periode sinyal 32 mili detik. Untuk mengubah periodik sinyal ini, program tinggal mengubah periode overflow timer.

Listing Program 8.11 Fungsi Mengubah Periodik Gelombang

```
void TIM_Change_Periodic(void){
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    HAL_TIM_Base_Stop(&htim6);
    HAL_TIM_Base_DeInit(&htim6);

    htim6.Instance = TIM6;
    htim6.Init.Prescaler = 0;
    htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim6.Init.Period = Tim_Periodic;
    htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
    {
        Error_Handler();
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig)
!= HAL_OK)
    {
        Error_Handler();
    }

    HAL_TIM_Base_Start(&htim6);
    DAC_Signal_Generator();

}
```

Pengubahan periode ini dilakukan dengan mengubah variabel *Tim_Periodic* melalui 2 buah saklar, yang masing-masing untuk menaikan dan menurunkan nilai *Tim_Periodic*. Setiap kali saklar ditekan, periodik diturunkan atau dinaikan 50.

Listing Program 8.12 Fungsi Pembacaan Saklar untuk Mengubah Periodik Gelombang

```
if(!HAL_GPIO_ReadPin(KEY2_GPIO_Port, KEY2_Pin)){
    keypresstimer=0;
    while(!HAL_GPIO_ReadPin(KEY2_GPIO_Port, KEY2_Pin)){
        keypresstimer++;
        HAL_Delay(1);
```

```

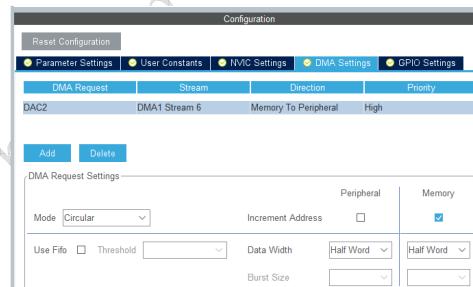
    }
    if(keypresstimer>10){
        Tim_Periodic+=50;
        TIM_Change_Periodic();
    }
}

if(!HAL_GPIO_ReadPin(KEY3_GPIO_Port, KEY3_Pin)){
    keypresstimer=0;
    while(!HAL_GPIO_ReadPin(KEY3_GPIO_Port, KEY3_Pin)){
        keypresstimer++;
        HAL_Delay(1);
    }
    if(keypresstimer>10){
        Tim_Periodic-=50;
        TIM_Change_Periodic();
    }
}

```

8.2.2 DMA MODE CIRCULAR

DAC dan timer bekerja seperti program sebelumnya, tetapi DMA bekerja dengan mode circular. Program untuk mode circular ini hampir sama dengan program untuk mode normal. Bedanya, fungsi *HAL_DAC_Start_DMA* tidak perlu dipanggil berulang-ulang melalui fungsi call back interupsi DMA. Karena pada saat DMA bekerja pada mode circular, ketika DMA telah selesai melakukan transfer data, secara otomatis DMA akan melakukan transfer lagi data yang sama. Pointer datanya secara otomatis akan kembali ke alamat awal. Seperti telah dijelaskan, mode circular sangat cocok untuk membangkitkan gelombang kontinyu seperti yang sedang di bahas ini.



Gambar 8.7 DAC dengan DMA Mode Circular

Oleh karena DAC akan bekerja secara kontinyu, maka untuk mengganti bentuk gelombang, DAC perlu dihentikan sementara dengan memanggil

fungsi *HAL_DAC_Stop_DMA*, baru kemudian di start kembali. Untuk mengganti frekuensi sama dengan program sebelumnya, tentu saja timer perlu dihentikan juga, karena timer juga bekerja secara kontinyu.

by Kang U2Man (u_2man@yahoo.co.id)

Bab 9

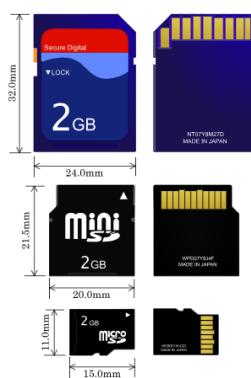
BERMAIN-MAIN DENGAN KARTU MIKRO SD DAN FATFS

Mikrokontroler merupakan perangkat dengan sumber atau memori terbatas. STM32F207 misalnya “hanya” diberi memori flash sebesar 1 Mbyte dan RAM 128 Kbyte. Walaupun memang 1 Mbyte itu sudah cukup besar untuk menyimpan aplikasi sistem embedded yang komplek. Namun ketika sistem embedded digunakan sebagai *logger* untuk menyimpan pembacaan sensor secara periodik misalnya, maka diperlukan tambahan memori eksternal. Kartu mikro SD (*Micro Secure Digital Card*) adalah salah satu contohnya, dan STM32F207 telah dilengkapi periperal untuk mengakses mikro SD, yaitu SDIO.

Untuk mengakses mikro SD bisa dilakukan dengan 2 cara, yaitu akses secara langsung atau *raw* dan menggunakan sistem file (*FAT file system*). Akses cara pertama dilakukan seperti halnya ketika mengakses EEPROM dengan mengakses data per alamat. Karena pada dasarnya, mikro SD terdiri atas byte-byte memori yang telah dialamati. Sedangkan untuk mengakses dengan menggunakan sistem file, diperlukan sebuah middleware untuk menangani sistem filenya, dalam hal ini sistem file FAT32. Middleware yang akan digunakan adalah sistem file yang didukung oleh STM32CubeMx yang bersifat open source yaitu FAT FS dari www.elm-chan.org (FatFs). Dan buku ini hanya akan membahas mengakses mikro SD dengan sistem file.

9.1 MIKRO SD

Selain berbentuk IC (chip), media penyimpanan juga dibuat dalam bentuk kartu. Di mulai dengan *PC Card* (PCMCIA), kemudian sejak 1994 berkembang *Compactflash*, *Smartmedia* dan *Miniature card*. Sejak PDA dan handphone berkembang yang membutuhkan kartu memori dengan ukuran lebih kecil, dibuatlah kartu MMC (*Multimedia Card*) dan kartu SD. Mikro SD adalah kartu SD dengan ukuran lebih kecil.



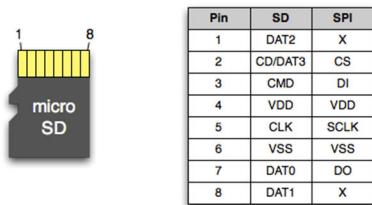
Gambar 9.1 SD, Mini SD dan Mikro SD

Kartu mikro SD, untuk selanjutnya akan disebut mikro SD, dirancang untuk memenuhi kebutuhan akan media penyimpanan dengan memperhatikan keamanan, kapasitas, dan performa untuk perangkat elektronik, terutama perangkat portabel atau mobil seperti handphoe dalam hal pengolahan file audio dan video. Selain itu mikro SD bersifat *non-volatile*, data akan tetap tersimpan walau pun catu daya tidak ada.

Secara default, mikro SD mempunyai kecepatan transfer data sampai 12.5 Mbyte per detik pada clock 25 MHz dan 4 bit data di tegangan 3,3V. Mikro SD keluaran terbaru kecepatan bahkan bisa mencapai 6.24 Gbps. Sedangkan kapasitas, mikro SD bisa berukuran 2 GB untuk mikro SD jenis SDSC (*Standard Capacity SD MemoryCard*), 2 GB – 32 GB untuk jenis SDHC (*High Capacity SD Memory Card*) dan 32 Gb – 2 TB untuk SDXC (*Extended Capacity SD Memory Card*).

Mikro SD mempunyai 8 pin untuk antarmukanya. Ada 2 mode yang bisa digunakan untuk mengakses mikro SD melalui pin-pin tersebut. Mode pertama menggunakan antarmuka SDIO dan mode kedua dengan menggunakan antarmuka SPI, mengingat tidak semua mikrokontroler

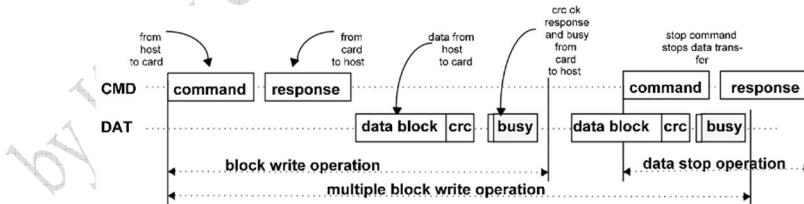
mempunyai periperal SDIO. Mode SDIO sendiri bisa menggunakan mode 4 bit atau mode 1 bit. Dan dalam pembahasan di buku ini, hanya akan menggunakan mode SDIO 4 bit.



Gambar 9.2 Pinout Mikro SD

Sebelum bisa diakses, mikro SD harus diinisialisasi terlebih dahulu. Inisialisasi terutama untuk menentukan mode antarmuka, SDIO atau SPI, menentukan fungsi dan mode kerja mikro SD. Saat pertama kali dinyalakan, mikro SD hanya bisa diakses melalui pin DAT0/DO. Inisialisasi kemudian menentukan lebar bit data yang akan digunakan. Inisialisasi juga menentukan ukuran blok data yang digunakan pada setiap transfer data, defaultnya 512 byte.

Protokol komunikasi dengan mikro SD melibatkan *command*, *response* dan data. Command merupakan byte-byte perintah yang dikirimkan dari host/mikrokontroler kepada mikro SD, sedangkan respond merupakan balasan dari mikro SD terhadap command yang dikirimkan. Dan data bisa berasal dari host atau pun dari mikro SD. Untuk validasi digunakan *checksum* (CRC). Proses transfer data biasanya dilakukan per blok, misal 512 byte, karena transfer data akan menjadi lebih cepat.



Gambar 9.3 Diagram Operasi Tulis Beberapa Blok

Di mode SDIO, command dan response dikirim secara serial melalui pin CMD dan data dikirim melalui pin DAT (1 bit atau 4 bit). Dengan demikian pin CMD dan DAT bersifat 2 arah. CMD dan DAT disinkronkan dengan sinyal clock yang dikirim dari host melalui pin CLK. Sehingga di sisi host, pin CLK adalah output dan di sisi mikro SD

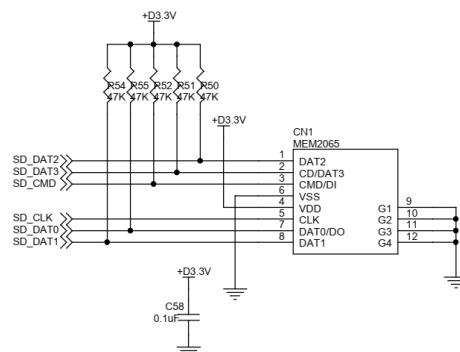
adalah input. Penjelasan lebih rinci mengenai protokol komunikasi mikro SD bisa dilihat di dokumentasi yang dikeluarkan oleh asosiasi SD (www.sdcard.org).

Oleh karena berbentuk kartu, untuk bisa dipasang di PCB mikro SD membutuhkan soket atau konektor. Berbagai jenis konektor banyak beredar dipasaran ada yang jenis *push pull* dengan atau tanpa fungsi deteksi kartu. Fungsi deteksi kartu pada dasarnya hanya penambahan saklar di konektor yang akan terhubung ke GND ketika kartu dimasukan.



Gambar 9.4 Berbagai Jenis Konektor Mikro SD

Pin-pin mikro SD bisa dihubungkan secara langsung dengan STM32F207, karena level digital untuk keduanya sama yaitu 3,3V. Jika menggunakan mikrokontroler dengan level digital 5V, maka diperlukan pengubah level tegangan agar tidak merusak kartu mikro SD. Lembaran data mikro SD merekomendasikan penambahan resistor pull up di semua pin mikro SD. Untuk kehandalan yang lebih tinggi mungkin juga diperlukan dioda anti listrik statik (ESD) dipasang di semua pin mikro SD.



Gambar 9.5 Skematik Mikro SD

Untuk komunikasi dengan mikro SD, STM32CubeMx telah menyediakan library HAL yang bisa digabungkan dengan middleware sistem file (FatFS). STM32F207 mendukung mode SDIO 1 bit dan 4 bit. DMA juga

bisa digunakan untuk mengakses mikro SD, terutama ketika aplikasi menggunakan RTOS, STM32CubeMx mengharuskan penggunaan DMA.

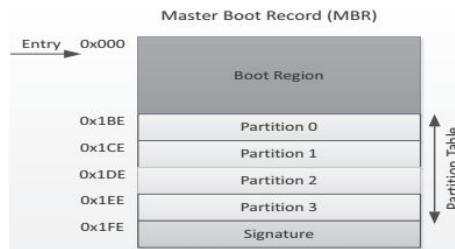
9.2 SISTEM FILE

Sistem file merupakan cara bagaimana data disimpan (tulis) dan diambil (baca) dalam sebuah media penyimpanan dengan tujuan untuk memudahkan dan mempercepat proses penulisan dan pembacaan data. Sistem file bisa diperlakukan dengan cara penyimpanan barang di gudang. Untuk memudahkan penyimpanan dan pengambilan kembali, barang tersebut diberi identitas (nama atau indeks) dan disimpan di rak/lemari yang juga tercatat lokasinya. Identitas dan lokasi barang tersebut disimpan dalam sebuah daftar (list). Jika akan mengambil sebuah barang, maka perlu melihat dulu ke daftar tersebut, di rak mana barang tersebut berada, kemudian, setelah lokasi diketahui, berjalan ke rak di mana barang tersebut berada. Di sistem file, ada bagian di memori yang digunakan untuk menyimpan lokasi di mana data berada.

Sistem file berkembang seiring dengan perkembangan teknologi komputer dan sistem operasi. Sistem file FAT (*File Allocation Table*) misalnya dimulai dengan versi FAT12 yang menggunakan 12 bit (4096) cluster dengan setiap cluster berukuran 8 KB sehingga kapasitas maksimum adalah 32 MB. Ketika teknologi berkembang, Microsoft selaku pengembang FAT membuat versi 16 bit dan kapasitas maksimum menjadi 2 GB. Kemudian dikembangkan lagi menjadi FAT32 dengan kapasitas sampai 8 TB (Tera Byte). Sekarang Microsoft di sistem operasinya (Windows) menggunakan sistem operasi NTFS (*New Technology File System*). Selain Microsoft, Apple juga mengembangkan sistem file untuk sistem operasinya (macOS dan iOS). APFS (*Apple File System*) dikenalkan pada tahun 2016 untuk menggantikan HFS+ (*Hierarchical File System*) yang telah digunakan sejak 1998. Sistem operasi Linux juga mempunyai sistem file sendiri yang dinamakan dengan ext (ext2, ext3 dan ext4). Selain itu XFS, JFS dan btrfs. Buku ini hanya akan membahas sistem file FAT, khususnya FAT32.

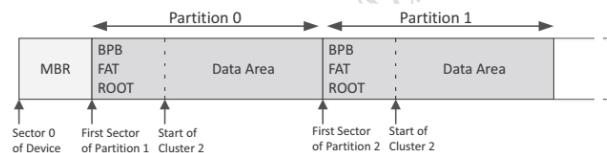
Dalam sistem file FAT32, media penyimpanan dibagi menjadi MBR (*Master Boot Record*) dan beberapa partisi (maksimum 4 partisi). MBR sendiri menempati sektor pertama dari media, 1 atau 2 sektor. MBR berisi *boot region* yang merupakan kode boot loader DOS yang ditulis ketika media pertama kali diformat. MBR juga berisi tabel partisi yang

menyimpan informasi setiap partisi, jenis partisi, sector awal dan akhir partisi, dan juga jumlah total sector setiap partisi. Jika jumlah sector ini bernilai 0, maka partisi tersebut belum diformat (masih tersedia untuk membuat partisi baru).



Gambar 9.6 Isi MBR

Di setiap partisi akan diawali dengan BPB (*BIOS Parameter Block*), diikuti oleh struktur FAT dan direktori root, baru kemudian area data yang menempati area terbanyak partisi. BPB berisi informasi mengenai jumlah byte per sektor, jumlah sektor per cluster, jumlah total sektor di dalam partisi dan jumlah entry di direktori root. Struktur FAT sendiri yang akan memberikan nama ke sistem file, dan menyimpan informasi mengenai sebuah cluster, apakah cluster tersebut berisi sebuah file, kosong atau bahkan rusak. Sementara direktori root bisa berisi file maupun sub-direktori-sub-direktori (sub-folder).



Gambar 9.7 Media dengan 2 Partisi

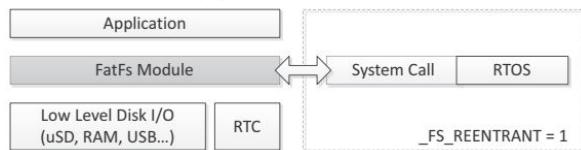
File sistem menggunakan penamaan untuk file dan direktori (folder). Untuk nama file, dikenal juga ekstensi file yang biasanya menggambarkan jenis dari file tersebut, apakah file berisi teks, gambar, audio atau video. Pada awalnya berkembang penamaan 8.3, artinya 8 karakter untuk nama file dan 3 karakter untuk ekstensinya, seperti yang dipakai di sistem operasi DOS. Standar 8.3 ini kemudian dikembangkan menjadi standar LFN (*Long File Name*) yang pertama kali diimplementasikan di sistem operasi *Windows NT* 3.5 tahun 1994. LFN mendukung penamaan file dan direktori sampai 255 karakter yang termasuk spasi dan karakter non alfa numerik (kecuali karakter-karakter yang telah digunakan oleh

COMMAND.COM di DOS yaitu \, /, *, ?, ", <, >, dan |). Panjang maksimum 255 karakter ini berlaku juga ketika memanggil file dengan nama direktorinya, artinya nama file dan nama direktorinya jika digabungkan juga maksimal 255 karakter.

9.3 FATFS

FatFs merupakan middleware yang dibuat untuk mengimplementasikan sistem file ke sistem embedded. FatFs dikembangkan untuk berjalan di mikrokontroler seperti 8051, AVR, PIC, ARM, Z80, RX dan lain-lain. Middleware ini dikembangkan oleh seorang insinyur sistem embedded yang tinggal di Jepang. FatFs mungkin middleware sistem file yang bersifat open yang paling banyak dipakai. STM32CubeMx sendiri telah mengintegrasikan FatFs sebagai salah satu middleware yang bisa digunakan untuk STM32. Saat buku ini ditulis, STM32CubeMx menggunakan FatFs versi R0.11 untuk STM32F207 dan R0.12c untuk STM32F7 dan STM32F4. Versi FatFs terbaru adalah R0.13 dan bisa diakses melalui link http://elm-chan.org/fsw/ff/00index_e.html.

FatFs merupakan sistem file yang kompatibel dengan sistem file FAT yang digunakan oleh sistem operasi Windows (FAT12, FAT16 dan FAT32) dan juga sistem file exFAT, termasuk sistem LFN. FatFs mendukung volume/drive jamak baik secara fisik (misal membaca flash disk USB dan mikro SD secara bersamaan) maupun partisi sampai 10 volume. Jumlah file yang dibuka secara bersamaan juga tidak terbatas tergantung memori yang tersedia di sistem. FatFs juga bisa digunakan secara aman (*thread safe*) dengan RTOS.



Gambar 9.8 Struktur FatFs

FatFs menyediakan fungsi antarmuka/API (*Application Programming Interface*) untuk berkomunikasi dengan aplikasi dan media penyimpanan, sehingga FatFs sangat mudah untuk dari satu aplikasi ke aplikasi lain atau dari media satu ke media yang lain. Ketika misal sebuah aplikasi dengan media penyimpanan mikro SD akan diganti dengan media penyimpanan memori flash SPI, maka yang perlu disesuaikan

hanyalah fungsi-fungsi yang mengakses ke media penyimpanan tersebut (*low level disk IO*). Aplikasi dan FatFs sendiri mungkin tidak perlu diubah.

Fungsi antarmuka dengan aplikasi dikelompokan sebagai berikut:

1. Fungsi API untuk akses file
 2. Fungsi API untuk akses direktori
 3. Fungsi API untuk manajemen file dan direktori
 4. Fungsi API untuk manajemen volume dan konfigurasi sistem.
- Fungsi API untuk akses file digunakan untuk operasi file, seperti membuka atau membuat file, menutup file, membaca atau menulis data ke file dan fungsi-fungsi lain yang berhubungan untuk mengakses file.

Tabel 9.1 Fungsi Akses File

Fungsi	Deskripsi
<i>f_open</i>	Fungsi untuk membuka atau membuat file
<i>f_close</i>	Fungsi untuk menutup file, jika sebelumnya ada fungsi penulisan, maka data yang ditulis akan disimpan ke media
<i>f_read</i>	Fungsi membaca data dari file
<i>f_write</i>	Fungsi menulis data ke file
<i>f_lseek</i>	Fungsi untuk memindahkan pointer data dari file yang dibuka
<i>f_truncate</i>	Fungsi untuk memotong ukuran file ke posisi pointer. Tidak berefek jika pointer sudah berada diakhir file
<i>f_sync</i>	Fungsi ini hampir sama dengan fungsi <i>f_close</i> tapi file masih terbuka dan bisa meneruskan fungsi baca, tulis atau memindahkan pointer data
<i>f_forward</i>	Fungsi untuk membaca data dari file kemudian meneruskannya ke sebuah <i>stream</i> tanpa melalui buffer data
<i>f_expand</i>	Fungsi untuk menambah blok data ke file
<i>f_gets</i>	Fungsi untuk membaca data <i>string</i> dari file sampai ketemu karakter '\n' (0x0A), 1 baris data <i>string</i>
<i>f_putc</i>	Fungsi untuk menulis 1 karakter ke file
<i>f_puts</i>	Fungsi untuk menulis 1 baris data <i>string</i> ke file
<i>f_printf</i>	Fungsi untuk menulis data terformat ke baris, seperti fungsi printf ke port serial
<i>f_tell</i>	Fungsi untuk membaca atau menulis pointer file
<i>f_eof</i>	Fungsi untuk mengecek apakah pointer data sudah berada di akhir file

<i>f_size</i>	Fungsi untuk membaca ukuran file
<i>f_error</i>	Fungsi untuk men-test error

Fungsi akses direktori, digunakan membuka atau menutup sebuah direktori, membaca nama file atau sub-direktori di direktori tersebut dan juga fungsi lain untuk mengakses sebuah direktori.

Tabel 9.2 Fungsi Akses Direktori

Fungsi	Deskripsi
<i>f_opendir</i>	Fungsi untuk membuka sebuah direktori
<i>f_closedir</i>	Funci untuk menutup direktori
<i>f_readdir</i>	Fungsi untuk membaca isi direktori
<i>f_findfirst</i>	Fungsi untuk membuka dan membaca isi pertama sebuah direktori
<i>f_findnext</i>	Fungsi untuk membaca item berikutnya dari sebuah direktori

Fungsi untuk manajemen file dan direktori merupakan fungsi yang bisa digunakan untuk akses file atau akses direktori. Misal fungsi untuk mengecek keberadaan sebuah file atau direktori, mengganti nama sebuah file atau direktori. Namun ada juga fungsi yang hanya bisa digunakan untuk direktori, seperti fungsi untuk membuat atau berganti direktori.

Tabel 8.1 Fungsi Manajemen File dan Direktori

Fungsi	Deskripsi
<i>f_stat</i>	Fungsi untuk mengecek keberadaan sebuah file atau direktori
<i>f_unlink</i>	Fungsi untuk menghapus file atau direktori
<i>f_rename</i>	Fungsi untuk mengganti nama atau memindahkan file atau direktori
<i>f_chmod</i>	Fungsi untuk mengubah atribut sebuah file atau direktori
<i>f_utime</i>	Fungsi untuk mengubah tanda waktu sebuah file atau direktori
<i>f_mkdir</i>	Fungsi untuk membuat sebuah direktori
<i>f_chdir</i>	Fungsi untuk mengubah direktori yang aktif
<i>f_chdrive</i>	Fungsi untuk mengubah drive yang aktif

<i>f_getcwd</i>	Fungsi untuk membaca drive atau direktori yang aktif
-----------------	--

Fungsi untuk manajemen volume dan konfigurasi sistem dipakai untuk mengatur media penyimpanan. Sebelum dilakukan operasi akses file atau direktori, sebuah media perlu di-*mount* terlebih dahulu bila tidak fungsi-fungsi file atau direktori tidak akan bisa digunakan. Fungsi konfigurasi sistem bisa juga digunakan untuk memberi atau membaca label ke sebuah disk drive atau memformat dan membuat partisi.

Tabel 8.2 Fungsi Manajemen Volum dan Konfigurasi

Fungsi	Deskripsi
<i>f_mount</i>	Fungsi untuk meregister volume menjadi area kerja bagi sistem file
<i>f_mkfs</i>	Fungsi untuk memformat drive
<i>f_fdisk</i>	Fungsi untuk mempartisi drive
<i>f_getfree</i>	Fungsi untuk membaca kapasitas total dan ruang kosong sebuah drive
<i>f_getlabel</i>	Fungsi untuk membaca label sebuah drive
<i>f_setlabel</i>	Fungsi untuk memberi label sebuah drive
<i>f_setcp</i>	

Untuk mengakses media penyimpanan, pustaka FatFs menyediakan fungsi-fungsi antarmuka dengan media penyimpanan tersebut. Pustaka FatFs sendiri tidak menyediakan isi dari fungsi-fungsi ini, hanya menyediakan fungsi prototipenya. Karena memang media penyimpanan bisa bermacam-macam, misal mikro SD, flash disk USB, memori flash SPI, memori flash dengan antarmuka paralel, hard disk, CD dan lain-lain, yang tentu saja cara mengaksesnya berbeda-beda. Programer harus menulis sendiri program untuk fungsi-fungsi ini, sesuai dengan media penyimpanan yang digunakan.

Pada saat aplikasi mengharuskan untuk mengganti media penyimpanan, misal dari mikro SD ke hard disk, maka mungkin hanya fungsi-fungsi ini yang harus ditulis ulang. Fungsi-fungsi untuk akses file dan lain-lain tidak perlu diubah, karena memang dirancang untuk tidak tergantung ke hardware atau platform tertentu.

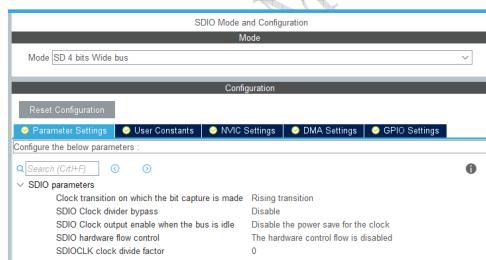
Tabel 8.3 Fungsi Antarmuka Media

Fungsi	Deskripsi
<i>disk_status</i>	Fungsi untuk membaca status dari media penyimpanan
<i>disk_initialize</i>	Fungsi untuk menginisialisasi media penyimpanan
<i>disk_read</i>	Fungsi untuk membaca data dari sektor media penyimpanan
<i>disk_write</i>	Fungsi untuk menulis data ke sektor media penyimpanan
<i>disk_ioctl</i>	Fungsi untuk kendali
<i>get_fattime</i>	Fungsi untuk membaca jam dan tanggal, misal dari RTC. Hasil pembacaan digunakan untuk digunakan sebagai jam dan tanggal file.

9.4 BERMAIN-MAIN DENGAN FATFS

9.4.1 OPERASI BACA/TULIS FILE

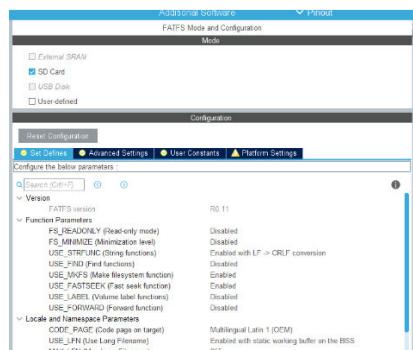
Pada contoh program ini, akan ditunjukkan bagaimana menggunakan koneksi SDIO untuk mengakses mikro SD dengan menggunakan sistem file FatFS. SDIO bekerja pada mode SD 4 bit, pengaturan SDIO mengikuti pengaturan STM32CubeMx, dengan interupsi dan DMA tidak diaktifkan. Port serial USART1 diaktifkan di baud rate 115200 untuk mengirim status program.



Gambar 9.9 Pengaturan SDIO

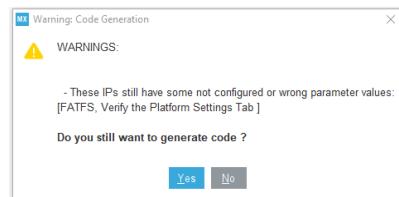
Sistem file FatFs diaktifkan dari tab *Middleware* dengan mode koneksi SD Card. FatFs yang didukung oleh STM32F207 adalah versi R0.11, untuk versi STM32 yang lain, misal STM32F4 atau STM32F7, mungkin menggunakan versi R0.12c. Di webnya sendiri, ketika buku ini ditulis, sudah mempunyai versi R.13c. Bisa saja program menggunakan versi FatFs terbaru, namun source code untuk FatFs versi terbaru perlu ditambahkan secara manual, dan mungkin masih harus mengedit secara

manual. Dalam contoh program di buku ini, fitur LFN (*long file name*) selalu diaktifkan, pengaturan lain mengikuti pengaturan STM32CubeMx.



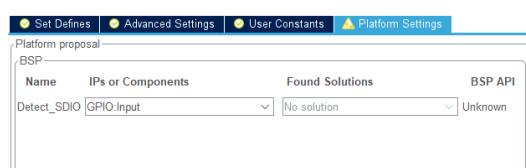
Gambar 9.10 Pengaturan FatFs

Ketika menyimpan STM32CubeMx mungkin akan memberikan pesan peringatan bahwa ada parameter yang mungkin belum diatur di tab *Platform Setting*. Peringatan ini bisa diabaikan.



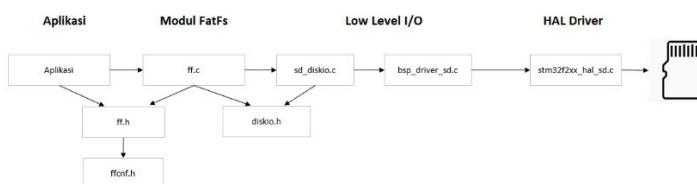
Gambar 9.11 Peringatan FatFs

Platform Setting pada dasarnya digunakan oleh FatFs untuk membaca status dari media (disk) yang digunakan. Konektor mikro SD mungkin mempunyai fungsi untuk mendeteksi ada atau tidak adanya mikro SD yang masuk di konektor. Fungsi ini biasanya menggunakan sebuah saklar yang mungkin terhubung saat ada mikro SD di dalam konektor dan terbuka saat tidak ada mikro SD, atau dengan kondisi sebaliknya. Jika saklar ini dihubungkan dan pin GPIO STM32F207, maka pin GPIO ini bisa diatur sebagai input dan dimasukan ke platform setting ini (*Found Solution*).



Gambar 9.12 Tab Platform Setting

STM32CubeMx kemudian akan membuat file-file source yang diperlukan untuk membuat aplikasi sistem file FatFs. Modul FatFs sendiri hanya terdiri atas 1 file source,yaitu file *ff.c*, selebihnya adalah file-file source untuk mengakses mikro SD. File *sd_diskio.c* merupakan file yang berisi untuk antarmuka ke media penyimpanan, fungsi untuk inisialisasi, baca, tulis dan membaca status media. Sedangkan file *bsp_driver_sd.c* merupakan file untuk mengakses mikro SD melalui driver HAL (*stm32fxx_hal_sd.c*). Programer tinggal mengembangkan di sisi aplikasi, file-file yang dihasilkan oleh STM32CubeMx pada dasarnya tidak perlu dimodifikasi.

**Gambar 9.13 Struktur File Source FatFs**

Jika *Platform Setting* diaktifkan, STM32CubeMx akan membuat file tambahan yaitu file *fatfs_platform.c*. File ini berisi fungsi *BSP_PlatformIsDetected* yang pada dasarnya membaca pin yang terhubung dengan fungsi deteksi mikro SD.

Listing Program 9.1 Fungsi Deteksi Mikro SD dengan GPIO

```

uint8_t BSP_PlatformIsDetected(void) {
    uint8_t status = SD_PRESENT;
    /* Check SD card detect pin */
    if(HAL_GPIO_ReadPin(SD_DETECT_GPIO_PORT, SD_DETECT_PIN) != GPIO_PIN_RESET)
    {
        status = SD_NOT_PRESENT;
    }
    /* USER CODE BEGIN 1 */
    /* user code can be inserted here */
    /* USER CODE END 1 */
    return status;
}
  
```

Fungsi ini akan dipanggil ketika menginisialisasi mikro SD, melalui fungsi *BSP_SD_IsDetected*. Ketika platform tidak digunakan fungsi *BSP_SD_IsDetected* akan langsung memberikan nilai kartu terdeteksi (*SD_PRESENT*).

Untuk memulai menggunakan FatFs, maka fungsi pertama yang harus dipanggil adalah *f_mount*. Fungsi ini pada dasarnya adalah untuk menginisialisasi media penyimpanan, dalam hal ini mikro SD. Jika mikro SD terdeteksi dan terinisialisasi, fungsi ini kemudian akan membaca sistem file yang tersimpan dalam mikro SD, informasi ini ditulis ketika mikro SD diformat. Tapi ini tergantung kepada parameter ketika fungsi *f_mount* dipanggil. Fungsi *f_mount* membutuhkan 3 parameter, pointer ke pada sistem file (tipe data *FATFS*), nomor drive yang akan di-mount, dan variabel *opt* yang merupakan pilihan, apakah media akan langsung di-mount (di-inisialisasi dan dibaca sistem file yang ada di media tersebut) atau tidak. Jika variabel *opt* bernilai 0, maka drive akan di-mount ketika ada fungsi yang dipanggil untuk mengakses file atau direktori, misal fungsi membuka file (*f_open*). Sedangkan jika *opt* bernilai 1, fungsi *f_mount* akan secara langsung untuk melakukan mount ke media tersebut. Ketika fungsi *f_mount* berhasil, ketika nilai kembali fungsi itu *FR_OK*, maka fungsi-fungsi yang lain siap digunakan.

Listing Program 9.2 Fungsi Baca/Tulis File

```
/* USER CODE BEGIN 2 */
printf("FatFS File Write/Read Example\r\n");
printf("Mount and read micro SD Card\r\n");
if(f_mount(&fs, (TCHAR const*)"",1) != FR_OK){
    printf("Micro SD not detected or not initialized\r\n");
}
else{
    printf("Micro SD OK\r\n");
    printf("Open and create file FileTest.txt\r\n");
    res = f_open(&TheFile, "FileTest.txt",
FA_CREATE_ALWAYS|FA_WRITE);
    if(res==FR_OK){
        printf("Open/Create File FileTest.txt OK\r\n");
        printf("Write text to file...\r\n");
        sprintf((char*)FileBuffer,"This text will be wrote to the
file");
        res=f_write(&TheFile,FileBuffer ,
strlen((char*)FileBuffer),(void*)&OperationCode);
        f_close(&TheFile);
        if(res==FR_OK){
            printf("Open file FileTest.txt\r\n");
            res=f_open(&TheFile,"FileTest.txt",FA_READ);
            if(res==FR_OK){
                printf("Read the file...\r\n");
                res=f_read(&TheFile, FileBuffer, 512,
(void*)OperationCode);
                if((res==FR_OK)&&(OperationCode>0)){
                    printf("Open file OK. Reading %d bytes\r\n",OperationCode);
                }
            }
        }
    }
}
```

```

        printf("File content
%s\r\n",FileBuffer);
    f_close(&TheFile);
}
}
}
else{
    printf("Open/Create File Error\r\n");
}
}

/* USER CODE END 2 */

```

Setelah fungsi *f_mount* berhasil, program kemudian akan membuka file dengan memanggil fungsi *f_open*. Fungsi ini membutuhkan 3 parameter, pointer file yang akan dibuka (variabel *TheFile* bertipe pointer *FIL*), pointer kepada nama file, dan mode pembacaan file. Ada beberapa mode yang bisa dipilih, seperti ditunjukkan tabel di bawah. Mode ini juga bisa di-OR-kan, seperti dalam contoh program ini yang meng-OR-kan mode *FA_CREATE_ALWAYS* dengan *FA_WRITE*. Pilihan mode ini artinya buka file dengan nama yang ditunjukan oleh parameter kedua, jika nama file itu belum ada maka akan dibuat file baru, sedangkan jika sudah ada maka file dengan nama yang sama akan ditimpa, dan akan dilakukan operasi penulisan ke file.

Tabel 9.3 Mode Operasi File

Mode	Keterangan
<i>FA_READ</i>	Membuka file untuk dibaca.
<i>FA_OPEN_EXISTING</i>	Membuka file yang sudah ada, jika tidak ada operasi baca akan gagal
<i>FA_WRITE</i>	Membuka file untuk ditulis
<i>FA_CREATE_NEW</i>	Membuat file baru, jika file sudah ada, maka fungsi akan gagal
<i>FA_CREATE_ALWAYS</i>	Membuat file baru, jika file sudah ada file tersebut akan ditimpa
<i>FA_OPEN_ALWAYS</i>	Membuka file, jika file tidak ada akan dibuat file baru

Ketika fungsi *f_open* berhasil, variabel *res* bernilai *FR_OK*, maka sebuah file baru dengan nama *FileTest.txt* terbentuk dan siap ditulis. Penulisan dilakukan dengan memanggil fungsi *f_write*, dengan parameter pointer kepada file yang bersangkutan saat operasi *f_open* (variabel *TheFile*), pointer kepada data yang akan ditulis, panjang data dan pointer kepada

sebuah variabel yang nantinya akan berisi berapa byte data yang berhasil ditulis. Oleh yang akan dibuat adalah file teks, maka isi data yang ditulis juga berupa data teks, "This text will be wrote to the file". Teks ini disimpan di variabel *FileBuffer* (data array 512 byte). Teks diisikan ke buffer dengan perintah standar C *sprintf*. Fungsi ini hampir sama dengan fungsi *printf* yang sudah dipelajari sebelumnya, hanya saja *sprintf* akan memformat data ke memori (buffer). Panjang data yang akan ditulis didapat dengan membaca panjang teks di buffer (fungsi *strlen*). Ketika operasi penulisan ini berhasil, maka res akan bernilai *FR_OK* dan variabel *OperationCode* akan bernilai sama dengan panjang data yang berhasil ditulis ke file. Jika *OperationCode* tidak sama dengan panjang data yang ada di variabel *FileBuffer*, dalam hal ini lebih kecil atau nol, maka itu menandakan bahwa media (mikro SD) sudah penuh. Perintah *f_close* kemudian akan memutup file *FileTest.txt* dan sekaligus menulisnya secara permanen ke mikro SD. Selama perintah *f_close* belum dipanggil, maka operasi *f_write* bisa terus dilakukan, sehingga program bisa menulis beberapa baris teks ke file.

Proses selanjutnya adalah proses untuk menguji bahwa pembutuan dan penulisan file telah berhasil. Ini dilakukan dengan membaca file *FileTest.txt* membaca isinya dan menampilkan isi tersebut ke port serial. Fungsi *f_open* kali ini menggunakan mode *FA_READ* karena hanya akan melakukan operasi pembacaan. Operasi pembacaan, fungsi *f_read*, parameternya sama dengan operasi penulisan, hanya saja nanti hasil pembacaan data akan disimpan di variabel *FileBuffer*. Dalam program, panjang data yang akan dibaca ditetapkan sebanyak 512 byte (sesuai dengan ukuran *FileBuffer*). Panjang data aktual yang terbaca akan disimpan di *OperationCode*, jika *OperationCode* bernilai 0 atau kurang dari 512, berarti data dalam file tersebut sudah habis terbaca (*end of file*). Selama belum EOF dan perintah *f_close* belum dipanggil, maka operasi baca juga bisa terus dilakukan. Bisakah file sekaligus dibaca? Misal ukuran file dibaca terlebih dahulu, lalu file tersebut dibaca sekaligus sesuai ukurannya. Jawabannya bisa, hanya saja program harus mengalokasikan memori (buffer) sebesar ukuran file. Semakin besar ukuran data yang dibaca/ditulis proses akses file akan semakin cepat. Data yang terbaca kemudian dikirimkan ke port serial.

Catatan, pada saat buku ini ditulis, STM32CubeIDE menggunakan versi 1.0.2. Versi ini kemungkinan ada *bug*, pada saat program dikompilasi dengan mode *debug* fungsi *f_mount* selalu gagal, proses inisialisasi mikro

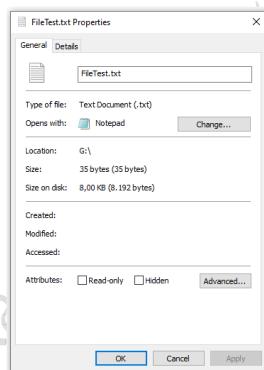
SD selalu gagal. Tetapi ketika dikompilasi dengan mode *release* program bisa berjalan baik.

```
FatFs File Write/Read Example
Mount and read micro SD Card
Micro SD OK
Open/Create file fileTest.txt
Open/Create File fileTest.txt OK
Write text to file...
Open file fileTest...
Read file fileTest...
Open file OK. Reading 35 bytes
File content This text will be wrote to the file
```

Gambar 9.14 Keluaran Program di Port Serial

9.4.2 JAM DAN TANGGAL AKSES FILE

Ketika file yang dihasilkan oleh contoh program sebelumnya dibaca propertinya, misal melalui *Windows Explorer*, bisa dilihat bahwa file tersebut tidak mempunyai tanda jam dan tanggal (*datetime stamp*), kapan file tersebut dibuat (*created*), dimodifikasi (*modified*) dan terakhir diakses (*accessed*). Walaupun FatFs sendiri mendukung untuk memberikan tanggal dan jam untuk file yang dibuat dengan fungsi *get_fattime* (ada di file *fatfs.c*), namun dalam contoh program tersebut fungsi itu tidak dimodifikasi.



Gambar 9.15 Atribut Jam dan Tanggal File tidak ada

Listing Program 9.3 Fungsi get_fattime

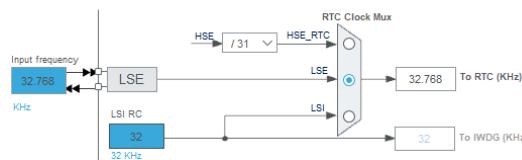
```
DWORD get_fattime(void)
{
    /* USER CODE BEGIN get_fattime */
    return 0;
    /* USER CODE END get_fattime */
}
```

Fungsi `get_fattime` pada dasarnya adalah fungsi dengan nilai kembali bertipe 32 bit (DWORD). Jam dan tanggal dikodekan sebagai berikut:

- Bit 31:25, berisi data tahun dari tahun 1980 (0...127), sebagai contoh 39 untuk tahun 2019.
- Bit 24:21, berisi data bulan (1...12)
- Bit 20:16, berisi data tanggal (1...31)
- Bit 15:11, berisi data jam (0...23)
- Bit 10:5, berisi data menit (0...59)
- Bit 4:0, berisi data detik/2, misalnya 25 untuk detik ke-50.

Untuk menggunakan fungsi `get_fattime`, tentu saja diperlukan data jam dan tanggal. Data tersebut bisa diperoleh dengan menggunakan sumber jam dan tanggal (kalender), salah satu sumber yang bisa digunakan adalah *Real Time Clock* (RTC). STM32F207 sudah mempunyai RTC internal, sehingga tidak diperlukan lagi RTC eksternal.

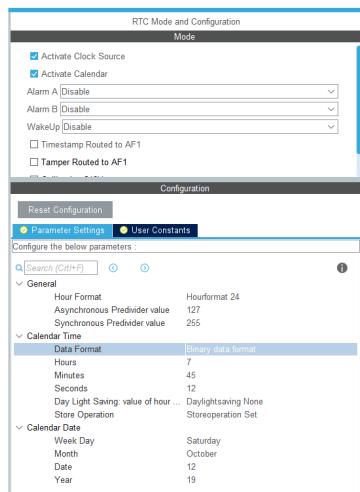
Seperti telah dijelaskan di bab sebelumnya, RTC bisa menggunakan clock internal (LSI) atau clock eksternal (LSE), bahkan bisa juga menggunakan clock dari HSE. Sumber clock dari LSE maupun HSE akan membuat RTC lebih akurat. Namun ketika menggunakan HSE, ketika catu daya utama mati, clock ke RTC akan terhenti walaupun pin VBAT dihubungkan dengan sebuah batere coin, mengingat VBAT tidak mencatu daya kepada osilator HSE. Untuk itu direkomendasikan menggunakan osilator LSE untuk akurasi dan agar kalender RTC tetap berjalan walaupun tegangan utama terputus, dengan catatan batere terhubung ke pin VBAT.



Gambar 9.16 Sumber Clock RTC

Untuk menggunakan RTC, dari STM32CubeMX cukup dengan mengaktifkan sumber clock RTC dan mengaktifkan fungsi kalender. Format jam bisa menggunakan format 12 jam atau 24 jam. Untuk format 12 jam digunakan tanda AM dan PM, yang bisa dibaca dari register RTC_TR bit ke-22. Kalender (jam dan tanggal) perlu diberikan nilai awal, nilai awal ini akan dipakai setiap kali reset. Dengan menggunakan register back up, maka kalender RTC tidak perlu diinisialisasi setiap kali

program reset, sehingga jam dan tanggal RTC tidak kembali ke nilai awal tersebut. Hal ini dilakukan oleh program seperti yang akan dijelaskan.



Gambar 9.17 Pengaturan RTC

Untuk membaca kalender RTC, pustaka HAL menyediakan 2 fungsi, *HAL_RTC_GetTime* untuk membaca jam, dan *HAL_RTC_GetDate* untuk membaca tanggal. Fungsi ini harus dipanggil setidaknya setiap 1 detik sekali, agar data RTC dibaca secara *real time*.

Dalam contoh program ini, setelah *mount* mikro SD berhasil, program akan membuat sebuah file dengan nama *FileTest1.txt*. Kemudian program melakukan pengulangan sebanyak 10 kali, membaca RTC dan menuliskan jam dan tanggal ke dalam file, dengan memanggil fungsi terformat *f_printf*. Setiap perulangan akan di-delay selama 1 detik, untuk memberikan nilai detik RTC berubah. Setelah penulisan berhasil dan menutup file, file tersebut kemudian dibuka kembali untuk dibaca dan ditampilkan ke port serial. Karena file berbentuk teks dan saat penulisan selalu diakhiri oleh karakter ganti baris '\r\n', maka pembacaan bisa menggunakan fungsi *f_gets*. Digabungkan dengan fungsi *f_eof*, selama pointer file masih belum diakhir file, maka program terus membaca file *f_gets* dan menampilkannya ke port serial. Fungsi *f_gets* akan memberikan nilai kembalian banyaknya karakter yang dibaca, jika lebih dari 0 maka data tersebut akan dikirim ke port serial.

Listing Program 9.4 File dengan Jam dan Tanggal

```
/* USER CODE BEGIN 2 */
printf("FatFS Jam dan Tanggal Akses File\r\n");
```

```

if(f_mount(&fs, (TCHAR const*)"",1) != FR_OK){
    printf("Mikro SD tidak terbaca\r\n");
    for(;;);
}
printf("Membuat File FileTest1.txt\r\n");
res = f_open(&TheFile, "FileTest1.txt", FA_CREATE_ALWAYS|FA_WRITE);
if(res!=FR_OK){
    printf("Open file gagal\r\n");
}
flag=1;
for(i=0;i<10;i++){
    HAL_RTC_GetTime(&hrtc, &RTCTime, RTC_FORMAT_BIN);
    HAL_RTC_GetDate(&hrtc, &RTCDate, RTC_FORMAT_BIN);
    res=f_printf(&TheFile, "Log %d: Jam %02d:%02d:%02d Tanggal:
%02d:%02d:%02d\r\n", i+1, RTCTime.Hours, RTCTime.Minutes,
RTCTime.Seconds, RTCDate.Date, RTCDate.Month, 2000 + RTCDate.Year);
    if(res==0){
        printf("Penulisan file gagal atau mikro SD penuh\r\n");
        flag=0;
        break;
    }
    HAL_Delay(1000);
}

f_close(&TheFile);

if(flag){
    printf("Membaca Isi File FileTest1.txt\r\n");
    res = f_open(&TheFile, "FileTest1.txt", FA_READ);
    if(res==FR_OK){
        while(!f_eof(&TheFile)){
            if(f_gets((char*)FileBuffer,512,&TheFile)>0){
                printf("%s\r\n",FileBuffer);
            }
        }
    }
    f_close(&TheFile);
}

/* USER CODE END 2 */

```

Untuk memberikan tanda tanggal dan jam ke file, fungsi *get_fattime* perlu dimodifikasi. Fungsi ini harus membaca jam dan tanggal dari RTC, kemudian mengubahnya ke format 32 bit seperti telah dijelaskan di atas. Perlu diketahui, data tahun yang terbaca dari RTC berukuran 8 bit (0-255), oleh karena itu perlu ditambahkan dengan 2000 (sesuai milenial sekarang). Data ini kemudian dikurangi dengan 1980, karena tahun di pustaka FatFs dimulai dari tahun 1980.

Listing Program 9.5 Fungsi get_fattime dengan RTC

```

DWORD get_fattime(void)
{
    /* USER CODE BEGIN get_fattime */
    RTC_TimeTypeDef RTCTime;
    RTC_DateTypeDef RTCDate;

    HAL_RTC_GetTime(&hrtc, &RTCTime, RTC_FORMAT_BIN);
    HAL_RTC_GetDate(&hrtc, &RTCDate, RTC_FORMAT_BIN);

    return ((DWORD)((2000+RTCDate.Year) - 1980) << 25)
        | ((DWORD)RTCDate.Month << 21)
        | ((DWORD)RTCDate.Date << 16)
        | ((DWORD)RTCTime.Hours << 11)
        | ((DWORD)RTCTime.Minutes << 5)
        | ((DWORD)RTCTime.Seconds >> 1);

    /* USER CODE END get_fattime */
}

```

Seperti telah dijelaskan di atas, bahwa fungsi back-up RTC tidak dibuat oleh STM32CubeMx, sehingga perlu diimplementasikan sendiri. Implementasi dilakukan dengan menggunakan register back-up. Register ini akan dicatu dari tegangan VBAT sehingga walaupun tegangan utama (VDD) terputus, nilai di register back-up ini akan tetap tersimpan, dengan catatan VBAT tersambung ke sebuah batere.

Fungsi implementasi dilakukan saat melakukan inisialisasi RTC (di fungsi *MX_RTC_Init* file *rtc.c*). Pertama program akan membaca sebuah register back-up (*RTC_BKP_DR0*) dan dibaca isinya. Jika menunjukan nilai tertentu, misal 0x32F2, ini berarti register back-up pernah ditulis sebelumnya. Ini berarti juga jam dan tanggal RTC pernah diinisialisasi, sehingga kalender RTC tidak perlu diinisialisasi kembali. Tetapi jika nilai register back-up tidak sama dengan 0x32F2, maka register back-up tersebut akan ditulisi dengan nilai 0x32F2 dan RTC akan diinisialisasi dengan jam dan tanggal yang telah ditentukan saat pembuatan program dengan STM32CubeMx.

Listing Program 9.6 Implementasi Fungsi Back-up

```

/* USER CODE BEGIN Check_RTC_BKUP */
if(HAL_RTCEx_BKUPRead(&hrtc, RTC_BKP_DR0) != 0x32F2){
    HAL_RTCEx_BKUPWrite(&hrtc,RTC_BKP_DR0,0x32F2);
}
else{
    return;
}

```

```
/* USER CODE END Check_RTC_BKUP */
```

Apakah RTC akan tetap berjalan ketika dicatu dari VBAT? Pada saat VDD terputus, seperti dijelaskan di bab sebelumnya, STM32F207 mempunyai mekanisme untuk men-switch dari VDD ke VBAT. Tegangan VBAT ini akan mencatu osilator OSC32K (dari LSI maupun LSE), RTC, logika wake-up, register back-up dan RAM back-up. Sehingga dengan adanya batere yang terhubung dengan VBAT, RTC akan tetap berjalan. Namun ketika sumber clock diambil dari HSE, maka ketika VDD terputus, RTC akan berhenti walaupun ada batere di VBAT. Karena memang VBAT tidak akan mencatu HSE. Ketika dinyalakan kembali (VDD dihubungkan kembali), maka jam dan tanggal RTC akan kembali ke jam dan tanggal RTC saat VDD terputus.

Dalam contoh program ini, RTC tidak disesuaikan (disinkronkan) dengan jam dan tanggal sebenarnya. RTC bisa disinkronkan dengan jam dan tanggal PC, misal melalui port serial. Atau bisa juga disinkronkan melalui internet dengan protokol NTP (*Network Time Protocol*). Untuk memperbarui jam dan tanggal RTC, HAL telah menyediakan fungsi *HAL_RTC_SetTime* untuk jam dan *HAL_RTC_SetDate* untuk tanggal.

9.4.3 MIKRO SD SEBAGAI PENGGANTI EEPROM

Seperti telah disebutkan di awal bab ini, bahwa mikro SD bersifat non-volatile. Oleh karena itu bisa digunakan untuk menyimpan data pengaturan, misal untuk menggantikan EEPROM untuk menyimpan pengaturan mode LED dari contoh bab sebelumnya. Data pengaturan dalam mikro SD disimpan dalam bentuk file, maka dapat dibuat atau disunting melalui PC. File ini biasanya berbentuk file teks (ASCII), sehingga bisa dengan mudah disunting dengan program pengolah teks, Notepad misalnya.

Contoh program kali ini akan menggunakan basis program pengendalian mode LED melalui port serial. Data pengaturan mode LED disimpan dalam bentuk file teks di dalam mikro SD. Masing-masing LED mempunyai file pengaturan tersendiri. Pengaturan mikro SD dan pustaka FatFs mengikuti pengaturan program sebelumnya, dengan RTC diaktifkan. LED dan port serial (USART1) juga mengikuti program LED sebelumnya, dengan menggunakan teknik interupsi tanpa DMA.

Bagian penting dari contoh program ini adalah bagian untuk membaca data pengaturan mode LED dari file dan penyimpanan mode pengaturan

mode LED ke dalam file. Pembacaan mode kerja dari file dilakukan di fungsi *Load_LED_Mode_FromFile* yang dipanggil melalui fungsi *LED_Default* sesaat setelah reset. Fungsi ini diawali dengan mengecek keberadaan folder/direktori *Setting* menggunakan fungsi *f_stat*. Jika folder tersebut belum ada, *f_stat* memberikan nilai *FR_NO_FILE*, maka program kemudian membuat folder *Setting* dengan fungsi *f_mkdir*.

Listing Program 9.7 Membaca Data Pengaturan dari File

```
void Load_LED_Mode_FromFile(void){
    FRESULT res;
    uint16_t i,j;
    uint8_t bfr[10];

    res = f_stat((char*)"Setting",&FileInfo);
    if(res==FR_NO_FILE){
        res=f_mkdir((char*)"Setting");
        if(res==FR_OK){
            printf("Membuat direktori berhasil\r\n");
        }
        else{
            printf("Membuat direktori gagal\r\n");
        }
    }
    else{
        printf("Direktori sudah ada\r\n");
    }

    for(i=0;i<5;i++){
        sprintf((char*)FileName,"Setting/LED%d_Config.txt",i+1);
        res=f_open(&SDFfile,(char*) FileName, FA_READ);
        if(res==FR_OK){
            res=f_read(&SDFfile, FileBuffer, 256, (void*)&FileOperation);
            if((res==FR_OK)&&(FileOperation>0)){
                //status#mode#blinkrate
                j=parsing_delimiter_string(bfr, FileBuffer, '#', 0,
strlen((char*)FileBuffer));
                LED[i].status=atoi((char*)bfr);
                j=parsing_delimiter_string(bfr, FileBuffer, '#', j,
strlen((char*)FileBuffer));
                LED[i].mode=atoi((char*)bfr);
                j=parsing_delimiter_string(bfr, FileBuffer, '#', j,
strlen((char*)FileBuffer));
                LED[i].blinkrate=atoi((char*)bfr);

            }
        }
        f_close(&SDFfile);
    }
}
```

Program selanjutnya membaca data pengaturan dengan membaca file pengaturan untuk setiap LED. File data pengaturan ini berada di folder *Setting* dan diberi nama sesuai dengan nomor LED, misal *LED1_Config.txt* untuk pengaturan LED 1. File konfigurasi ini adalah file teks dengan format penyimpanan *status#mode#blinkrate*, misal jika file *LED1_Config.txt* berisi 1#1#100, artinya status LED tersebut adalah nyala (*status=1*), mode kerja berkedip (*mode=1*) dengan kecepatan kedip 100 mili detik (*blinkrate=100*). Isi file ini dibaca dengan fungsi *f_read* dan kemudian di-parsing seperti parsing data yang diterima dari port serial, melalui fungsi *parsing_delimiter_string*. Jika file konfigurasi untuk sebuah LED tidak ditemukan, maka LED akan bekerja dengan mode default, seperti yang didefinisikan di fungsi *LED_Default*.

Ketika data pengaturan mode LED dikirim dari PC, data ini kemudian disimpan di dalam file, fungsi *Save_LED_Mode_ToFile*. Penyimpanan data menggunakan fungsi penyimpanan data teks ke file dengan format (*f_printf*). Penyimpanan file ini disesuaikan dengan indeks LED yang bersangkutan yang disimpan di variabel *LedIndex*.

Listing Program 9.8 Penyimpanan Mode Kerja LED ke File

```
void Save_LED_Mode_ToFile(uint8_t LedIndex){  
    FRESULT res;  
  
    sprintf((char*)FileName,"Setting/LED%d_Config.txt",LedIndex+1);  
    res=f_open(&SDFile,(char*) FileName, FA_WRITE|FA_CREATE_ALWAYS);  
    if(res==FR_OK){  
        f_printf(&SDFile,"%d%d%d",LED[LedIndex].status,  
        LED[LedIndex].mode, LED[LedIndex].blinkrate);  
    }  
  
    f_close(&SDFile);  
}
```

Dari contoh pembacaan dan penulisan file yang ada diprogram ini, terlihat nama file, yang disimpan di variabel *FileName*, disatukan dengan nama foldernya. Hal ini dimungkinkan dengan mengaktifkan fitur LFN, dengan fitur ini pemanggilan nama file dan sub-foldernya bisa sampai dengan 255 karakter. Selain itu, pemanggilan nama file disatukan dengan nama foldernya, memungkinkan untuk memanggil file dari folder utama atau dari folder yang lain, sehingga tidak perlu masuk ke folder di mana file tersebut berada. Jika diinginkan memanggil nama file tanpa nama folder, maka harus masuk ke folder di mana file tersebut berada, jika tidak fungsi *f_open* akan menghasilkan error.

by Kang U2Man (u_2man@yahoo.co.id)

by Kang U2Man (u_2man@yahoo.co.id)

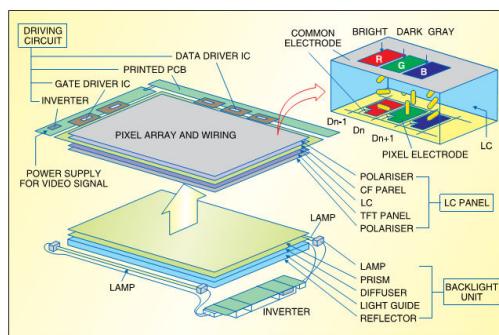
Bab 10

BERMAIN-MAIN DENGAN LCD TFT + TOUCH PANEL

Dalam alat sistem embedded, salah satu alat untuk menampilkan hasil dari sebuah proses atau keluaran data adalah LCD (*Liquid Crystal Display*). Salah satu jenis LCD adalah LCD TFT (*Thin Film Transistor*). LCD ini telah banyak digunakan sebagai penampil data pada banyak peralatan sistem embedded. Sebuah laptop juga menggunakan LCD jenis ini sebagai layar monitornya dengan resolusi *Full High Definition* (resolusi piksel 1920x1080). Untuk kelas mikrokontroler, mungkin hanya bisa mengendalikan LCD TFT resolusi rendah, mungkin sekelas VGA (800x600). Tapi dengan digabungkannya layar sentuh (*touch panel*), yang merupakan peralatan input, LCD TFT ini bisa digunakan untuk membuat aplikasi antarmuka antara manusia dengan mesin atau HMI (*Human Machine Interface*).

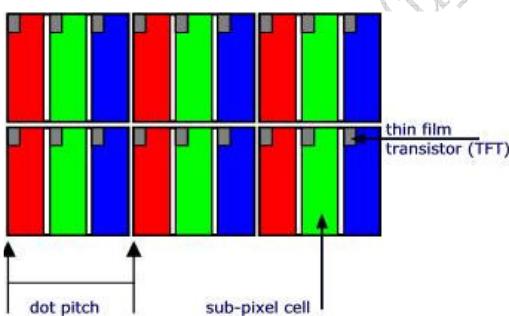
10.1 PRINSIP KERJA LCD TFT

Struktur sebuah LCD TFT ditunjukkan oleh gambar 10.1. LCD TFT pada dasarnya terdiri atas LCD modul itu sendiri, yang tentu saja terbuat dari TFT yang membentuk susunan titik (piksel) sesuai resolusinya. Kemudian ada driver yang akan mengendalikan kerja LCD modul dan sebagai antarmuka dengan prosesor (MCU). LCD juga membutuhkan sebuah lampu *backlight* agar LCD bisa terlihat menampilkan data. Lampu backlight ini bisa menggunakan LED atau lampu FL (*Fluorecent lamp*). Lampu FL menggunakan tegangan tinggi sehingga dibutuhkan inverter untuk menyalakannya.



Gambar 10.1 Struktur LCD TFT

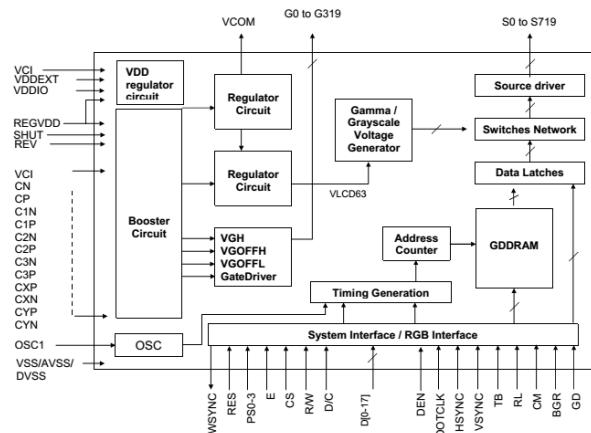
TFT pada dasarnya adalah sebuah transistor efek medan (*Field Effect Transistor*). Di LCD, TFT ini berperan aktif untuk mengontrol setiap piksel yang dimiliki oleh LCD, sehingga sering disebut *active matrix TFT*. Setiap piksel ini memiliki kemampuan untuk memancarkan warna yang sesuai. Untuk itu diperlukan backlight yang biasanya sebuah lampu flourescent atau LED warna putih. Setiap piksel terdiri atas 3 sub-piksel dengan warna merah, hijau dan biru (*Red, Green, Blue*). Menurut teori warna, dengan mengatur komposisi ketiga warna ini, maka warna-warna lain bisa dihasilkan. Setiap sub-piksel ini dikendalikan oleh sebuah TFT.



Gambar 9.18 Struktur Pikel LCD TFT

Warna-warna dihasilkan dengan mengatur tegangan atau arus atau bias ke setiap TFT. Dan tentu saja, diperlukan sebuah driver untuk mengendalikan TFT-TFT tersebut. Prosesor sepertinya tidak bisa mendrive secara langsung, karena pasti jumlah pin yang diperlukan tidak tersedia. Sebagai contoh sebuah LCD dengan resolusi 320×240 akan ada 960×240 piksel atau TFT yang diperlukan. IC driver ini dirancang agar mudah dihubungkan dengan mikrokontroler, pada umumnya menggunakan antarmuka paralel untuk kecepatan tinggi atau serial

(UART, SPI atau I2C) untuk menghemat pin, tentu saja kecepatan transfer data yang berkurang. Dalam pembahasan buku ini, LCD TFT yang digunakan adalah LCD dengan driver SSD1289 dari *Solomon Systech*.



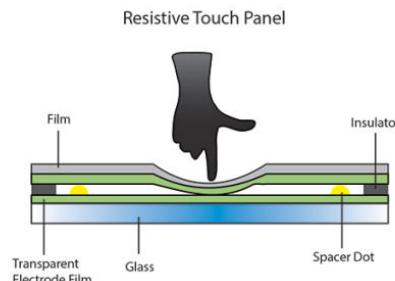
Gambar 10.2 Blok Diagram SSD1289

Sebuah driver TFT akan mempunyai sebuah regulator tegangan (*DC-DC Converter*) untuk menyediakan tegangan untuk membias TFT di panel LCD. GDDRAM (*Graphic Display Data RAM*) dan tentu saja driver untuk dihubungkan dengan panel LCD. IC driver biasanya dirancang untuk resolusi dan warna tertentu, misalnya SSD1289 dirancang untuk mendrive LCD 320×240 piksel dengan resolusi warna sampai 262K. Antarmukanya menggunakan data paralel 18 bit (agar bisa mendukung resolusi 262K). Namun bisa juga dikonfigurasi untuk antarmuka 8/9/16/18-bit atau SPI. Di dalamnya sudah terdapat GDDRAM 172,800 bytes ($240 \times 320 \times 18 / 8$), melalui GDDRAM inilah mikrokontroler menggambar sesuatu di panel LCD. Fitur-fiturnya diatur melalui register-register kendali. Register-register ini biasanya diinisialisasi saat pertama kali LCD dinyalakan.

Dengan mode paralel, mode 6800 atau 8080, bisa diakses sebagai RAM dengan kecepatan tinggi, mikrokontroler dengan fitur FSMC (*Flexible Static Memory Controller*) bisa memanfaatkan fitur ini. Sedangkan mode serial, digunakan untuk mikrokontroler dengan jumlah pin terbatas. Sinyal-sinyal sinkronisasi (VSYNC, HSYNC, DOTCLK, DEN) bisa digunakan untuk aplikasi animasi atau bahkan video.

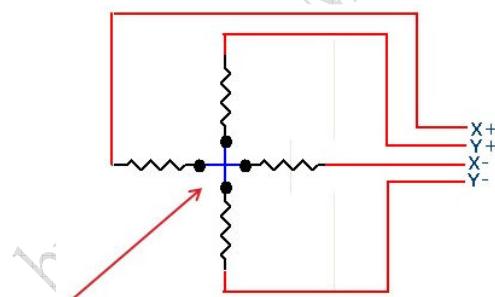
10.2 PRINSIP KERJA LAYAR SENTUH

Layar sentuh biasanya dipasang di atas LCD. Ada banyak jenis layar sentuh, seperti layar sentuh resistif, kapasitif, *surface acoustic wave*, layar sentuh optik dan lain-lain. Dalam buku ini digunakan layar sentuh resistif.



Gambar 10.3 Struktur Layar Sentuh Resistif

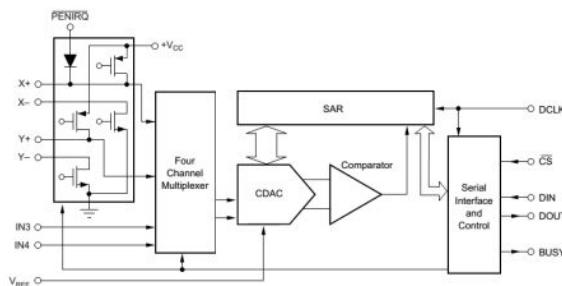
Secara sederhana, sebuah layar sentuh resistif bekerja ketika panel disentuh, maka akan mengubah resistansinya. Layar sentuh akan mempunyai 4 pin, dua untuk resistansi sumbu X ($X+$ dan $X-$) dan dua untuk resistansi sumbu Y ($Y+$ dan $Y-$). Sehingga layar sentuh akan mengeluarkan 2 nilai resistansi, resistansi yang diukur dari pin X dan nilai resistansi yang diukur dari pin Y. Dengan demikian, nilai-nilai resistansi ini akan menunjukkan posisi di mana layar sentuh ini di sentuh.



Gambar 10.4 Rangkaian Ekivalen Sebuah Layar Sentuh

Karena keluarannya analog, sebenarnya layar sentuh bisa dibaca langsung oleh mikrokontroler melalui input analog (ADC), tentu saja dengan rangkaian bias tertentu. Tapi kebanyakan, layar sentuh juga dibaca melalui sebuah driver. Dan module LCD TFT biasanya juga sudah termasuk driver untuk membaca layar sentuh ini. Driver ini bisa

dihubungkan dengan mikrokontroler secara serial (SPI atau mungkin I2C).



Gambar 10.5 Blok Diagram Driver Layar Sentuh

Pada dasarnya sebuah driver layar sentuh, misal ADS7843 atau XPT2046, merupakan sebuah ADC. Driver ini akan memberikan bias (tegangan) kepada resistansi sumbu X dan resistansi sumbu Y secara bergantian. Dengan adanya tegangan bias, akan menghasilkan tegangan di kedua resistansi tersebut. Jika driver membaca bahwa layar telah disentuh, driver bisa memberikan interupsi ke mikrokontroler, sehingga mikrokontroler kemudian membaca datanya melalui antarmuka yang telah disediakan (SPI).

Pembacaan koordinat layar sentuh tidak selamanya presisi. Ada beberapa hal yang mempengaruhi akurasi pembacaan ini, yang paling besar pengaruhnya adalah tegangan noise, faktor skala (*scaling factor*) dan pemasangan yang tidak pas dengan LCD. Tegangan noise bisa berasal dari panel TFT atau lampu backlight, terutama saat menggunakan lampu FL. Untuk itu, pembacaan layar sentuh resistif membutuhkan kalibrasi. Proses kalibrasi ini biasanya dilakukan dengan menekan titik-titik tertentu yang ditampilkan di LCD. Nilai pembacaan ini kemudian akan disimpan sebagai offset agar pembacaan sesuai dengan posisi piksel di LCD.

10.3 ANTARMUKA LCD DENGAN STM32F207

10.4.1 ANTARMUKA

Sebuah LCD TFT yang sudah dikemas menjadi sebuah modul, akan mempunyai konfigurasi sebagai berikut:

1. Port data 16 bit (DB0 – DB15), port data ini bersifat 2 arah (*bidirectional*). Dengan labar data 16 bit, LCD ini mendukung resolusi warna 65K.
2. Port kendali, yang terdiri atas kendali baca/tulis (RD/WR), pemilih LCD (CS) dan pin untuk memilih akses ke register/perintah atau RAM/Data (RS). Kemudian ada pin reset LCD (RSET).
3. Pin catu daya (VCC dab GND), untuk menyediakan tegangan untuk IC driver TFT, layar sentuh dan mungkin untuk MMC/SD. Selain itu ada juga pin untuk LED backlight (LEDA dan LEDK). Catau daya untuk LCD biasanya 5V, tegangan ini kemudian diturunkan menjadi 3,3V oleh sebuah IC regulator.

Untuk koneksi ke layar sentuh, mempunyai konfigurasi sebagai berikut:

1. Port SPI (MISO/OUT, MOSI/IN, CLK/DCLK, dan TP_CS).
2. Pin IRQ, yang aktif rendah untuk menunjukkan bahwa ada perubahan data di layar sentuh.
3. Pin BUSY, menunjukkan status dari IC driver layar sentuh.

LCD juga mungkin mempunyai soket MMC/SD yang antarmukanya menggunakan SPI. Dalam pembahasan buku ini, MMC/SD yang ada di LCD tidak akan digunakan.

LCD TFT320	
1	NC
2	LED _A
3	NC
4	RST
5	NC
6	CS
7	DB15
8	DB14
9	DB13
10	DB12
11	DB11
12	DB10
13	DB9
14	DB8
15	RD
16	WR
17	RS
18	NC
19	VCC
20	GND
21	DB0
22	DB1
23	DB2
24	DB3
25	DB4
26	DB5
27	DB6
28	DB7
29	TP_CLK
30	TP_CS
31	TP_SI
32	TP_BUSY
33	TP_SO
34	TP_JR0
35	SD_MISO
36	SD_SCK
37	SD_MOSI
38	SD_CS
39	NC
40	NC

Gambar 10.6 Konfigurasi LCD TFT

Dalam standar aliansi MIPI (*Mobile Industri Processor Interface*), konfigurasi seperti ini termasuk kategori MIPI-DBI (*Display Bus Interface*). Aliansi MIPI merupakan organisasi global yang mendefinisikan dan mempromosikan spesifikasi antarmuka untuk perangkat mobil (*smart phone*). Salah satunya antarmuka ke perangkat display, baik yang sudah ada maupun antarmuka baru.

MIPI-DBI mempunyai 3 jenis antarmuka:

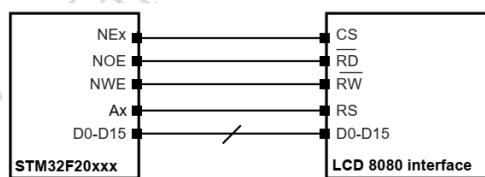
1. Jenis A, antarmuka berdasarkan bus Motorola 6800
2. Jenis B, antarmuka berdasarkan bus Intel 8080
3. Jenis C, antarmuka melalui protokol SPI

SSD1289, mendukung ketiga jenis antarmuka tersebut. Tetapi ketika IC driver tersebut sudah dipasangkan dengan modul TFT, mungkin hanya salah satu jenis antarmuka yang dipakai, dan LCD modul yang digunakan dalam pembahasan dibuku ini menggunakan antarmuka jenis B (Intel 8080).

Antarmuka Intel 8080 bisa dipandang sebagai konfigurasi memori sehingga antarmuka ini bisa diakses melalui perangkat FSMC. STM32F207 mempunyai FSMC dengan 4 bank memori, setiap bank memori berukuran masing-masing 256 MB (4x64 MB). Sinyal-sinyal kendali yang dimiliki oleh FSMC cocok dengan antarmuka LCD. Sinyal-sinyal tersebut adalah:

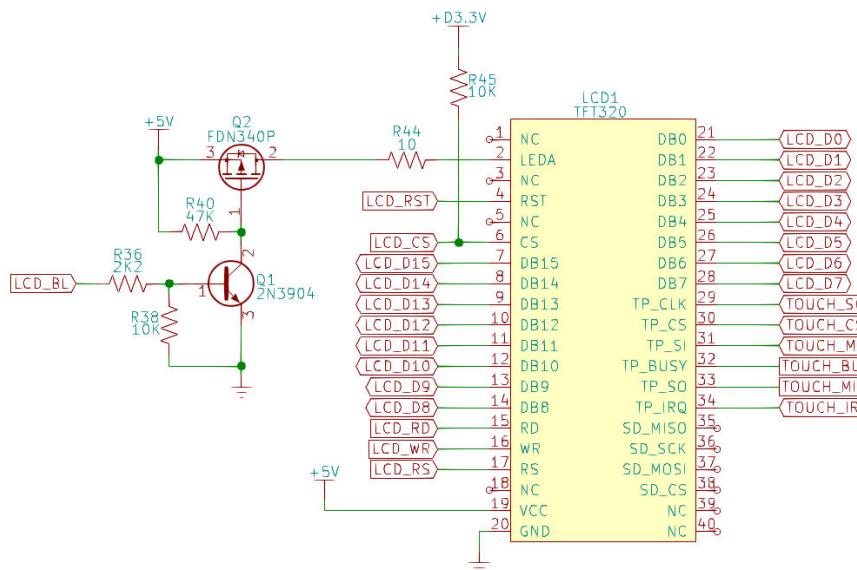
1. FSMC[D0:D15], bus data 16 bit
2. FSMC NEx, pemilih FSMC (CS), dengan x 1 – 4 (4 buah CS)
3. FSMC NOE, sinyal baca FSMC
4. FSMC NEW, sinyal tulis FSMC
5. FSMC Ax, bus alamat 26 bit

Antarmuka LCD dengan FSMC ditunjukkan oleh gambar 10.7, bus data dari LCD terhubung dengan bus data FSMC. Kendali baca/tulis LCD terhubung dengan kendali baca/tulis FSMC. Pin CS LCD dikendalikan oleh pin FSMC NEx, dan pin RS dihubungkan dengan sinyal Ax. Sinyal Ax ini menentukan alamat dari LCD (alamat register dan alamat RAM). Sinyal Ax bisa dipilih antara A0 – A25. Kendali lain dari LCD, seperti pin reset bisa menggunakan GPIO biasa. Untuk LED backlight juga bisa dikendalikan melalui GPIO atau dengan sinyal PWM.



NEx where x can be 1 to 4
Ax where x can be 0 to 25

Gambar 10.7 Antarmuka LCD dengan FSMC



Gambar 10.8 Contoh Skematik Antarmuka LCD TFT

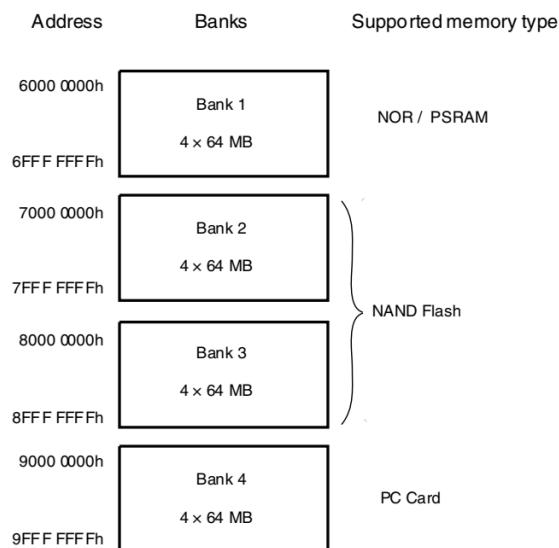
10.3.1 PENGALAMATAN

Alamat LCD akan ditentukan berdasarkan bank memori yang dipilih. Dari 4 bank memori yang dimiliki oleh STM32F207, hanya bank memori 1 yang bisa digunakan untuk antarmuka dengan LCD. Sehingga LCD akan berada di alamat antara 0x60000000 – 0x6FFFFFFF. Bank memori ini juga dibagi menjadi 4 bagian, masing-masing 64 MB. Pemilihan ini dilakukan dengan memilih pin FSMC NE. Ketika mengakses memori eksternal, sebuah register alamat 32 bit yang dinamakan HADDR akan menterjemahkan alamat yang dihasilkan oleh jalur alamat bus AHB ke jalar alamat memori eksternal.

Pemilihan pin FSMC NE ditentukan oleh register HADDR[27:26] sebagai berikut:

1. A27:A26 = 00, Bank 1 kendali FSMC NE1
2. A27:A26 = 01, Bank 1 kendali FSMC NE2
3. A27:A26 = 10, Bank 1 kendali FSMC NE3
4. A27:A26 = 11, Bank 1 kendali FSMC NE4

Dengan memilih FSMC NE3, alamat LCD akan berada di 0x68000000.



Gambar 10.9 Bank Memori STM32F207

Walaupun LCD akan diakses sebagai memori eksternal, LCD hanya akan menggunakan 2 alamat (pin RS), yaitu saat mengakses register (perintah) dan saat mengakses GDDRAM (data). Oleh karena itu hanya dibutuhkan 1 pin alamat untuk mengakses LCD ini. Walaupun register HADDR berukuran 32 bit, tetapi alamat yang akan diterjemahkan ke bus alamat eksternal hanya 26 bit (HADDR[25:0]). Oleh karena itu pin alamat yang bisa dipilih untuk kendali pin RS LCD adalah A0 – A25. Register HADDR pada dasarnya adalah alamat per byte, sementara memori dialami sebagai word, maka alamat aktual yang dikeluarkan ke bus alamat eksternal tergantung dengan lebar data (8 bit atau 16 bit). Ketika lebar data 16 bit, maka bank memori 64 MB akan diakses sebagai $64\text{ MB}/2 \times 16$. Oleh karena itu ketika digunakan lebar data 16 bit (seperti LCD ini) alamat yang dikeluarkan oleh register HADDR adalah HADDR[25:1]>>1.

Sebagai contoh ketika pin LCD RS dihubungkan dengan FSMC A16, maka bit register HADDR yang mempengaruhi adalah bit A17. Dengan demikian, untuk mengakses register LCD (RS=0) maka bit A17 harus bernilai 1, sehingga alamat akan berada di 0x68000000, dan untuk mengakses GDDRAM (RS=1) akan beralamat di 0x68020000.

A31	A30	A29	A28	A27	A26	A25	A24	A23	A22	A21	A20	A19	A18	A17	A16
0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	0
6				8				0						2	

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0				0				0				0		0	

Register HADDR

FSMC NEx ---> Bit A27:A26

FSMC A16 ---> Bit A17

Gambar 10.10 Register HADDR

10.3.2 PEWAKTUAN

Bagian terpenting lain yang harus diperhatikan adalah pewaktuan. Setiap perangkat memori, termasuk LCD, mempunyai batas pewaktuan sendiri, waktu operasi baca, tulis dan sebagainya. Parameter pewaktuan ini harus sama antara LCD dengan kendali FSMC yang dikendalikan oleh MCU. Jika pewaktuan FSMC melebihi kemampuan pewaktuan LCD, maka LCD tidak akan bisa diakses. Sebaliknya, jika pewaktuan FSMC diatur jauh di bawah kemampuan pewaktuan LCD, akses LCD tidak akan optimum. Misal LCD terlihat lambat ketika menampilkan sebuah gambar. Kecepatan piksel per detiknya akan turun. Sehingga pewaktuan FSMC harus diatur semaksimal mungkin dengan pewaktuan LCD.

Sebuah catatan aplikasi yang dikeluarkan oleh ST Microelectronics yang berjudul *TFT LCD interfacing with the high-density STM32F10xxx FSMC* ada 3 parameter yang harus diperhatikan ketika mengatur perangkat FSMC untuk komunikasi dengan LCD TFT,

1. Waktu setup alamat (*address setup time*)
2. Waktu penahan alamat (*address hold time*)
3. Waktu setup data (*data setup time*)

Di dokumen tersebut juga ditunjukkan bagaimana melakukan penghitungan pewaktuan dengan LCD yang digunakan dalam board evaluasi yang dibuat oleh ST. Namun sayangnya ketika dokumen tersebut dirujuk ke lembaran data dan manual referensi sangat sulit untuk mencocokkan perhitungan pewaktuan tersebut. Sehingga di buku ini pewaktuan optimum diperoleh dengan cara *try and error* seperti akan dijelaskan kemudian.

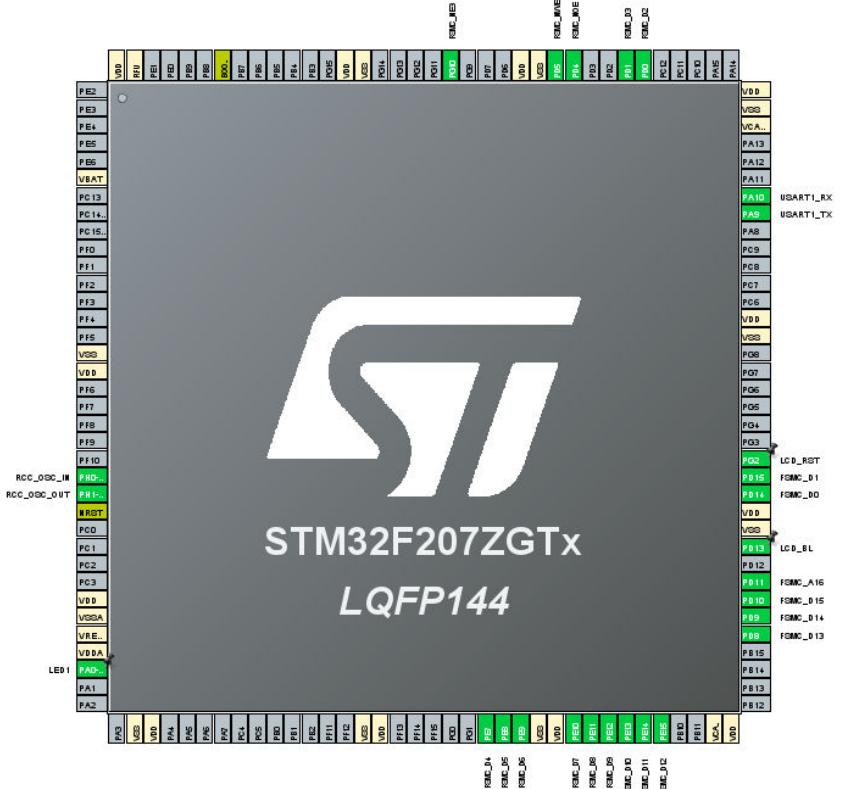
10.4 BERMAIN-MAIN DENGAN LCD

10.4.2 LCD – HELLO WORLD

Seperti telah dijelaskan, untuk antarmuka dengan LCD digunakan FSMC yang dikonfigurasi untuk antarmuka dengan memori NOR Flash/PSRAM/SRAM/ROM/LCD 1. *Chip Select* dipilih NE3, jenis memori LCD *interface* dan *LCD Register Select A16*, yang akan menempatkan LCD di alamat 0x68000000 untuk mengakses register LCD dan 0x68020000 untuk GDDRAM. Lebar data adalah 16 bit. Alamat ini dalam bahasa C dideklarasikan sebagai berikut:

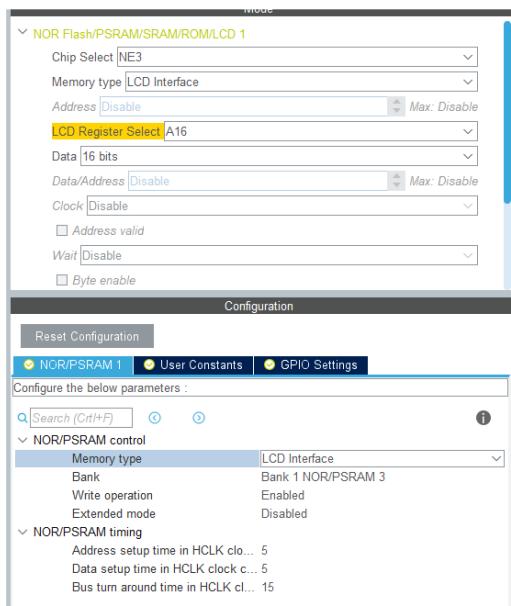
Listing Program 10.1 Pendeklarasian Alamat LCD

```
#define LCD_REG      (*(volatile unsigned short *) 0x68000000) /*  
RS = 0 */  
#define LCD_RAM      (*(volatile unsigned short *) 0x68020000) /*  
RS = 1 */
```



Gambar 10.11 Konfigurasi Pin FSMC untuk LCD

Selain itu diperlukan juga GPIO untuk mengendalikan pin reset LCD dan lampu backlight LCD. Pin reset LCD biasanya aktif rendah, logika LOW di pin reset akan mereset LCD dan logika HIGH akan mengaktifkan LCD. Untuk lampu backlight, yang biasanya digunakan MOSFET atau transistor bipolar, bisa menggunakan metode GPIO (logika HIGH atau LOW) atau dengan mode PWM. Jika digunakan mode PWM, pin GPIO yang digunakan harus mendukung fungsi PWM. Dengan PWM, kecerahan lampu backlight bisa diatur. Di dalam buku ini, hanya digunakan mode GPIO.



Gambar 10.12 Pengaturan FSMC untuk LCD

Untuk pewaktuan yang perlu dikonfigurasi ketika FSMC dihubungkan dengan LCD adalah *Address setup time*, *Data setup time* dan *Bus turn arround time*. Satuan ketiga parameter tersebut adalah siklus clock HCLK, artinya dengan frekuensi HCLK 120 MHz maka 1 siklusnya adalah 8,3 nano detik. Dari sini bisa diketahui bahwa *Address setup time* adalah 41,5 nano detik ($5 \times 8,3$). Pengaturan FSMC ini akan menghasilkan fungsi `MX_FSMC_Init` sebagai berikut.

Listing Program 10.2 Fungsi Inisialisai FSMC

```
void MX_FSMC_Init(void)
{
    FSMC_NORSRAM_TimingTypeDef Timing = {0};
```

```
/** Perform the SRAM1 memory initialization sequence
*/
hsram1.Instance = FSMC_NORSRAM_DEVICE;
hsram1.Extended = FSMC_NORSRAM_EXTENDED_DEVICE;
/* hsram1.Init */
hsram1.Init.NSBank = FSMC_NORSRAM_BANK3;
hsram1.Init.DataAddressMux = FSMC_DATA_ADDRESS_MUX_DISABLE;
hsram1.Init.MemoryType = FSMC_MEMORY_TYPE_SRAM;
hsram1.Init.MemoryDataWidth = FSMC_NORSRAM_MEM_BUS_WIDTH_16;
hsram1.Init.BurstAccessMode = FSMC_BURST_ACCESS_MODE_DISABLE;
hsram1.Init.WaitSignalPolarity = FSMC_WAIT_SIGNAL_POLARITY_LOW;
hsram1.Init.WrapMode = FSMC_WRAP_MODE_DISABLE;
hsram1.Init.WaitSignalActive = FSMC_WAIT_TIMING_BEFORE_WS;
hsram1.Init.WriteOperation = FSMC_WRITE_OPERATION_ENABLE;
hsram1.Init.WaitSignal = FSMC_WAIT_SIGNAL_DISABLE;
hsram1.Init.ExtendedMode = FSMC_EXTENDED_MODE_DISABLE;
hsram1.Init.AsynchronousWait = FSMC_ASYNCHRONOUS_WAIT_DISABLE;
hsram1.Init.WriteBurst = FSMC_WRITE_BURST_DISABLE;
/* Timing */
Timing.AddressSetupTime = 5;
Timing.AddressHoldTime = 15;
Timing.DataSetupTime = 5;
Timing.BusTurnAroundDuration = 15;
Timing.CLKDivision = 16;
Timing.DataLatency = 17;
Timing.AccessMode = FSMC_ACCESS_MODE_A;
/* ExtTiming */

if (HAL_SRAM_Init(&hsram1, &Timing, NULL) != HAL_OK)
{
    Error_Handler( );
}

}
```

Dari fungsi ini, ada beberapa parameter pewaktuan yang tidak bisa diatur melalui STM32CubeMx, yaitu *AddressHoldTime*, *CLKDivision*, dan *DataLatency*. Parameter yang, setelah dicoba, berpengaruh terhadap LCD adalah *AddressHoldTime* dan pada frekuensi HCLK 120 MHz nilainya minimal 15 siklus HCLK.

Untuk menguji apakah konfigurasi pewaktuan FSMC sudah sesuai dengan pewaktuan LCD adalah dengan cara mencoba membaca isi register *Device Code Read* yang beralamat di 0x00. Ketika isi register ini dibaca akan memberikan nilai 0x8989, yang merupakan ID dari IC driver SSD1289. Untuk membaca dari register SSD1289, maka yang diperlukan adalah menunjukkan alamat register yang akan dibaca melalui LCD_REG,

kemudian membacanya melalui LCD_RAM. Begitu juga ketika akan menulis ke register, alamat register disimpan di LCD_REG dan data disimpan di LCD_RAM. LCD_REG dan LCD_RAM, seperti telah dijelaskan, merupakan pointer ke alamat eksternal sesuai pengaturan FSMC. Pembacaan dan penulisan register dilakukan melalui fungsi:

Listing Program 10.3 Fungsi Penulisan dan Pembacaan Register LCD

```
void ssd1289_WriteReg(uint8_t LCDReg, uint16_t LCDRegValue){
    LCD_REG = LCDReg;
    LCD_RAM = LCDRegValue;
}

uint16_t ssd1289_ReadReg(uint8_t LCDReg){
    LCD_REG = LCDReg;
    return LCD_RAM;
}
```

Sehingga untuk membaca ID dari driver SSD1289, bisa dilakukan dengan fungsi:

Listing Program 10.4 Fungsi Pembacaan ID

```
uint16_t ssd1289_ReadID(void){
    return (ssd1289_ReadReg(0x00));
}
```

Setelah ID bisa dibaca, maka proses inisialisasi bisa dilakukan, tentu saja setelah pin reset diberi logika HIGH.

Dalam contoh buku ini untuk mengakses LCD digunakan 2 file sumber, lcd.c dan ssd1289.c serta ditambah beberapa file sumber yang berisi data font untuk menuliskan huruf dan angka. File lcd.c berisi fungsi-fungsi diperlukan untuk menggambar ke LCD, seperti menggambar garis, lingkaran atau menampilkan angka dan huruf. Sedangkan file ssd1289.c digunakan untuk melakukan fungsi-fungsi yang memang khusus untuk IC SSD1289. Tujuan dari pemisahan ini adalah, agar ketika diperlukan penggantian LCD yang berbeda drivernya, misal dengan IC IL9341, cukup membuat file, misal dengan nama il9341.c, sedangkan file lcd.c cukup membuat penyesuaian di fungsi BSP_LCD_Init.

Listing Program 10.5 Fungsi Inisialisasi LCD

```
uint8_t BSP_LCD_Init(void){
    uint8_t ret = LCD_ERROR;

    //Turn On LCD Backlight
```

```

HAL_GPIO_WritePin(LCD_BL_GPIO_Port, LCD_BL_Pin, GPIO_PIN_SET);
//LCD Reset
HAL_GPIO_WritePin(LCD_RST_GPIO_Port, LCD_RST_Pin, GPIO_PIN_SET);
//HAL_Delay(2);
HAL_GPIO_WritePin(LCD_RST_GPIO_Port, LCD_RST_Pin, GPIO_PIN_RESET);
HAL_Delay(2);
HAL_GPIO_WritePin(LCD_RST_GPIO_Port, LCD_RST_Pin, GPIO_PIN_SET);

HAL_Delay(10);

/* Default value for draw property */
DrawProp.BackColor = 0xFFFF;
DrawProp.pFont = &Font24;
DrawProp.TextColor = 0x0000;

if(ssd1289_drv.ReadID() == SSD1289_ID)
{
    lcd_drv = &ssd1289_drv;

    /* LCD Init */
    lcd_drv->Init();

    /* Initialize the font */
    BSP_LCD_SetFont(&LCD_DEFAULT_FONT);

    ret = LCD_OK;
}

return ret;
}

```

Fungsi-fungsi yang ada di lcd.c menggunakan sebuah pointer untuk mengakses ke driver LCD. Pointer ini adalah *lcd_drv* yang bertipe *LCD_DrvTypeDef*, yang berisi pointer ke fungsi-fungsi driver LCD.

Listing Program 10.6 Struktur LCD_DrvTypeDef

```

typedef struct
{
    void (*Init)(void);
    uint16_t (*ReadID)(void);
    void (*DisplayOn)(void);
    void (*DisplayOff)(void);
    void (*SetCursor)(uint16_t, uint16_t);
    void (*WritePixel)(uint16_t, uint16_t, uint16_t);
    uint16_t (*ReadPixel)(uint16_t, uint16_t);

    /* Optimized operation */
    void (*SetDisplayWindow)(uint16_t, uint16_t, uint16_t, uint16_t);
    void (*DrawHLine)(uint16_t, uint16_t, uint16_t, uint16_t);
    void (*DrawVLine)(uint16_t, uint16_t, uint16_t, uint16_t);

    uint16_t (*GetLcdPixelWidth)(void);
    uint16_t (*GetLcdPixelHeight)(void);
    void (*DrawBitmap)(uint16_t, uint16_t, uint8_t* );
}

```

```

    void (*DrawRGBImage)(uint16_t, uint16_t, uint16_t, uint16_t, uint8_t*);
}LCD_DrvTypeDef;

```

Dalam file `ssd1289.c` ada variabel yang juga bertipe `LCD_DrvTypeDef` yaitu `ssd1289_drv` yang langsung ke fungsi-fungsi untuk mengakses ke driver SSD1289.

Listing Program 10.7 Variabel `ssd1289_drv`

```

LCD_DrvTypeDef ssd1289_drv =
{
    ssd1289_Init,
    ssd1289_ReadID,
    ssd1289_DisplayOn,
    ssd1289_DisplayOff,
    ssd1289_SetCursor,
    ssd1289_WritePixel,
    ssd1289_ReadPixel,
    ssd1289_SetDisplayWindow,
    ssd1289_DrawHLine,
    ssd1289_DrawVLine,
    ssd1289_GetLcdPixelWidth,
    ssd1289_GetLcdPixelHeight,
    ssd1289_DrawBitmap,
    ssd1289_DrawRGBImage,
};

```

Dalam fungsi `BSD_LCD_Init`, ketika fungsi `ssd1289_drv.ReadID` memberikan nilai `0x8989` ini artinya LCD dengan driver SSD1289 telah terdeteksi, untuk itu variabel `lcd_drv` akan diberikan nilai pointer ke variabel `ssd1289_drv`. Artinya setiap pemanggilan fungsi dari `lcd_drv` akan diarahkan ke `ssd1289_drv`. Sebagai contoh ketika fungsi `lcd_drv->Init` dipanggil maka akan diarahkan ke fungsi `ssd1289_Init`. Jadi ketika driver LCD berganti, misal menjadi IL9341, tinggal membuat variabel `il9341_drv` kemudian mendefinisikan fungsi-fungsi yang sesuai dengan IL9341.

Kembali ke fungsi inisialisasi LCD. Proses inisialisasi pada dasarnya adalah memprogram register-register yang ada di SSD1289. Melalui register-register ini, SSD1289 diatur agar mode kerjanya sesuai dengan LCD yang ditanganinya dan juga mode antarmuka dengan prosesor. Sebenarnya secara hardware SSD1289 juga harus disesuaikan, tetapi itu sudah dilakukan oleh pembuat LCD modul. Sayangnya sangat sulit untuk mendapatkan skematik LCD modul sehingga pengaturan hardware SSD1289 tidak bisa dibahas di sini.

Register-register yang dimiliki oleh SSD1289 cukup banyak, tidak memungkinkan untuk dijelaskan satu-persatu. Register-register tersebut

akan mengatur osilator, kendali output, pengaturan warna dan sebagainya. Semua register berukuran 16 bit. Sebagai contoh register 0x00 (register osilator), ketika dibaca maka akan memberikan ID dari SSD1289 (0x8989). Tetapi register ini berfungsi juga mengatur osilator SSD1289. Pengaturan osilator dilakukan melalui bit OSCEN (bit ke-0), menset bit ini akan mengaktifkan osilator. Sehingga untuk mengaktifkan osilator register ini harus diberi nilai 0x0001.

Oscillator (R00h) (POR = 0000h)																	
R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	OSCEN
POR	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Gambar 10.13 Register 0x00

Contoh lain, register 0x11 (*Entry Mode*) akan mengatur sinkronisasi display dengan sinyal VSYNC (*Vertical Synchronization*) melalui bit VSMODE. Namun dalam kebanyakan modul LCD, pin VSYNC tidak dikeluarkan, sehingga sinkronisasi data display akan disesuaikan dengan clock internal SSD1289 (VSMODE=0). Ini juga akan meringankan kerja prosesor host, karena tidak perlu menyediakan sinyal VSYNC. Walaupun sinyal VSYNC akan sangat berguna untuk menampilkan gambar bergerak (film), karena dengan VSYNC eksternal pengiriman data akan bisa lebih cepat dari kecepatan sinkronisasi internal. Sehingga pergantian data setiap frame tidak akan menimbulkan *flicker*.

Entry Mode (R11h) (POR = 6830h)																	
R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	VSMODE	DFM1	DFM0	TRANS	OEDef	WMode	DMode1	DMode0	TY1	TY0	ID1	ID0	AM	LG2	LG1	LG0
POR	0	1	1	0	1	0	0	0	0	0	0	1	1	0	0	0	

Gambar 10.14 Register 0x11

Register ini juga akan mengatur mode warna, apakah akan mendukung warna 16 bit (65 ribu warna) atau 18 bit warna (262 ribu warna). LCD modul yang digunakan dalam buku ini hanya mendukung antarmuka 16 bit, tetapi dengan pengaturan melalui bit TY[1:0] antarmuka 16 bit juga bisa mendukung mode warna 18 bit. Data RGB bisa dikirim 2 atau 3 kali, tentu saja hal ini akan mengurangi kecepatan data. Dalam contoh buku ini hanya akan digunakan mode warna 16 bit.

Dengan memanfaatkan bit OEDef dan Dmode[1:0] bisa mengaktifkan fungsi *picture-in-picture* menampilkan gambar di atas gambar. Bit TRANS akan mengatur fungsi transparansi, sehingga gambar yang ditampilkan akan terlihat transparan dengan gambar yang ada di belakang (latar belakang).

Bit ID[1:0] dan bit AM akan mengatur penghitung alamat GDDRAM horisontal dan vertikal apakah akan dikurangi atau ditambah setiap kali dilakukan pengiriman data display. Sebagai contoh ketika ID[1:0] bernilai 00 dan AM=0 maka penunjuk alamat akan dikurangi baik horisontal maupun vertikal. Hal ini bisa digunakan untuk mengirimkan data ke area tertentu di LCD tanpa perlu menset posisi kursor secara berulang-ulang.

	ID[1:0] = "00" Horizontal: decrement Vertical: decrement	ID[1:0] = "01" Horizontal: increment Vertical: decrement	ID[1:0] = "10" Horizontal: decrement Vertical: increment	ID[1:0] = "11" Horizontal: increment Vertical: increment
AM = "0" Horizontal	00,00h 	00,00h 	00,00h 	00,00h
AM = "1" Vertical	00,00h 	00,00h 	00,00h 	00,00h

Gambar 10.15 Bit ID[1:0] dan Bit AM

Jika diinginkan akurasi warna, SSD1289 menyediakan fungsi pengaturan *gamma*. Ada 10 register untuk pengaturan ini. Pengaturan gamma membutuhkan pengukuran. LCD harus menampilkan pola warna tertentu kemudian diukur untuk mendapatkan nilai *chroma* tertentu (proses kalibrasi warna). Register gamma diatur agar nilai chroma yang diukur sesuai dengan spesifikasi yang telah ditentukan. Nilai register yang didapat kemudian disimpan di EEPROM. Setiap kali inisialisasi SSD1289, nilai dari EEPROM dibaca kembali untuk dikirimkan ke register gamma.

Register 0x4E, penghitung alamat GDDRAM X (horisontal), dan register 0x4F, penghitung alamat GDDRAM Y (vertikal) digunakan untuk menentukan kursor LCD. Register-register ini akan menentukan posisi awal dari data yang dikirimkan. Dengan memanipulasi register-register ini, bisa digunakan untuk mode *portrait* atau *landscape*. Dalam pembahasan buku ini mode yang digunakan adalah mode landscape.

Register-register lain bisa dilihat di lembaran data SSD1289.

Listing Program 10.8 Fungsi Inisialisasi SSD1289

```
void ssd1289_Init(void){
```

```
    ssd1289_WriteReg(0x0000,0x0001);
```

```
ssd1289_WriteReg(0x0001,0x3B3F); // 0x2B3F
ssd1289_WriteReg(0x0002,0x0600);
ssd1289_WriteReg(0x0003,0xA0A8);
ssd1289_WriteReg(0x0005,0x0000);
ssd1289_WriteReg(0x0006,0x0000);
ssd1289_WriteReg(0x0007,0x0233);
ssd1289_WriteReg(0x000B,0x0000);
ssd1289_WriteReg(0x000C,0x0000);
ssd1289_WriteReg(0x000D,0x000F);
ssd1289_WriteReg(0x000E,0x2B00);
ssd1289_WriteReg(0x000F,0x0000);
ssd1289_WriteReg(0x0010,0x0000);
ssd1289_WriteReg(0x0011,0x6018); // Up-Right
ssd1289_WriteReg(0x0016,0xEF1C);
ssd1289_WriteReg(0x0017,0x0003);
ssd1289_WriteReg(0x001E,0x00B8);
ssd1289_WriteReg(0x0023,0x0000);
ssd1289_WriteReg(0x0024,0x0000);
ssd1289_WriteReg(0x0025,0x8000);
ssd1289_WriteReg(0x0030,0x0707);
ssd1289_WriteReg(0x0031,0x0204);
ssd1289_WriteReg(0x0032,0x0204);
ssd1289_WriteReg(0x0033,0x0502);
ssd1289_WriteReg(0x0034,0x0507);
ssd1289_WriteReg(0x0035,0x0204);
ssd1289_WriteReg(0x0036,0x0204);
ssd1289_WriteReg(0x0037,0x0502);
ssd1289_WriteReg(0x003A,0x0302);
ssd1289_WriteReg(0x003B,0x0302);
ssd1289_WriteReg(0x0041,0x0000);
ssd1289_WriteReg(0x0042,0x0000);
ssd1289_WriteReg(0x0044,0xEF00);
ssd1289_WriteReg(0x0045,0x0000);
ssd1289_WriteReg(0x0046,0x013F);
ssd1289_WriteReg(0x0048,0x0000);
ssd1289_WriteReg(0x0049,0x013F);
ssd1289_WriteReg(0x004A,0x0000);
ssd1289_WriteReg(0x004B,0x0000);
ssd1289_WriteReg(0x004e,0);
ssd1289_WriteReg(0x004f,0);

}
```

Setelah IC driver diinisialisasi, LCD bisa segera untuk digambari, dengan cara mengirim data ke GDDRAM. Sebelum bisa mengirim data ke GDDRAM, SSD1289 perlu diberi perintah terlebih dahulu. Proses ini hampir sama dengan saat menulis ke register SSD1289, bedanya setelah alamat register didefinisikan, dalam hal ini register 0x22, maka data yang dikirim melalui LCD_RAM akan disimpan di GDDRAM, dan selama

tidak perintah ke LCD_REG, penulisan GDDRAM bisa terus dilakukan. Bersamaan dengan itu LCD akan menampilkan gambar sesuai dengan data yang dikirim.

Contoh pertama pemrograman LCD TFT ini akan menampilkan tulisan *HELLO WORLD!* di baris atas dengan tulisan, kira-kira ditengah LCD. Selain itu juga akan menampilkan bendera merah putih tepat di tengah layar LCD.



Gambar 10.16 Program Hello World LCD

Sebelum mulai pembahasan bagaimana menggambar ke layar LCD, ada baiknya untuk memahami bagaimana data dikirim ke driver LCD. seperti telah dijelaskan bahwa SSD1289 mendukung 18 bit data (262 ribu warna), namun LCD modul yang digunakan hanya mengeluarkan data 16 bit, sehingga SSD1289 harus diinisialisasi untuk bekerja di 16 bit (65 ribu warna). Walaupun sebenarnya dengan antarmuka 16 bit pun bisa mendukung 262 ribu warna, dengan catatan data dikirim lebih dari 2 sampai 3 kali.

Untuk antarmuka data 16 bit, SSD1289 mengharuskan bit ke-0 (D0) dan bit ke-9 (D9) tidak digunakan. Ini artinya D0 di konektor LCD modul sebenarnya mungkin akan terhubung dengan D1 di SSD1289, D8 akan terhubung dengan D10 dan seterusnya, sehingga SSD1289 tetap membaca 18 bit data, dengan D0 dan D9 diabaikan.

		Hardware pins																	
Interface	Cycle	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
18 bits		IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	x	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0	x
16 bits		IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8		IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0	
9 bits	1 st										IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	x
	2 nd										IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0	x
8 bits	1 st										IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	
	2 nd										IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0	

Remark : x Don't care bits
 Not connected pins

Gambar 10.17 Koneksi Pin Data untuk Berbagai Antarmuka Data

Dengan antarmuka 16 bit, data warna dipetakan dengan format 5-6-5, 5 bit untuk warna merah (*Red*), 6 bit untuk warna hijau (*Green*) dan 5 bit untuk warna biru (*Blue*). Namun urutan data, apakah RGB atau BGR, ditentukan oleh register 0x01 yaitu bit RL, bit BGR dan bit TB. Dengan modul LCD yang digunakan dan program pengaturan seperti ditunjukkan oleh fungsi *ssd1289_Init()* urutan warnanya adalah RGB. Sedangkan komposisi bit untuk warna dalam data 16 bit ditunjukkan oleh gambar di bawah.

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0
R					G					B					

Gambar 10.18 Format Data RGB

Sehingga definisi untuk warna merah, hijau dan biru akan menjadi seperti listing program di bawah. Definisi warna yang lain bisa dibuat dengan meng-komposisi-kan bit-bit RGB dengan nilai-nilai yang berbeda.

Listing Program 10.9 Definisi Warna RGB

```
#define LCD_COLOR_BLUE          0x001F
#define LCD_COLOR_GREEN         0x07E0
#define LCD_COLOR_RED           0xF800
```

Dalam contoh program pertama ini, setelah diinisialisasi LCD akan dibersihkan dengan memanggil fungsi *LCD_Clear* dengan warna hitam, kemudian menulikan kata “HELLO WORLD!” dan menggambar bendera merah putih.

Listing Program 10.10 Program Hello World LCD

```
/* USER CODE BEGIN 2 */
if (BSP_LCD_Init()==LCD_OK){
    BSP_LCD_Clear(LCD_COLOR_BLACK);
    BSP_LCD_SetBackColor(LCD_COLOR_BLACK);
    BSP_LCD_SetTextColor(LCD_COLOR_YELLOW);
    BSP_LCD_DisplayStringAt(0, 0, (uint8_t*)"HELLO WORLD!",
    CENTER_MODE);
    BSP_LCD_SetTextColor(LCD_COLOR_RED);
    BSP_LCD_FillRect(60, 60, 200, 60);
    BSP_LCD_SetTextColor(LCD_COLOR_WHITE);
    BSP_LCD_FillRect(60, 120, 200, 60);
}
/* USER CODE END 2 */
```

Fungsi *BSP_LCD_Clear()* pada dasarnya hanya menggambar garis horisontal dari baris pertama sampai baris 240 (dalam buku ini layar LCD

berorientasi *landscape* dengan resolusi 320 × 240) dengan warna yang ada di parameter fungsi tersebut, yaitu hitam. Atau dengan kata lain mengisi semua alamat GDDRAM dengan data 0x0000.

Listing Program 10.11 Fungsi BSP_LCD_Clear

```
void BSP_LCD_Clear(uint16_t Color){
    uint32_t counter = 0;
    uint32_t color_backup = DrawProp.TextColor;
    DrawProp.TextColor = Color;

    for(counter = 0; counter < BSP_LCD_GetYSize(); counter++)
    {
        BSP_LCD_DrawHLine(0, counter, BSP_LCDGetXSize());
    }
    DrawProp.TextColor = color_backup;
    BSP_LCD_SetTextColor(DrawProp.TextColor);

}
```

Untuk menggambar bendera merah putih dilakukan dengan menggambar kotak yang terisi penuh (fungsi *BSP_LCD_FillRect*). Fungsi ini hampir sama dengan fungsi *BSP_LCD_Clear*, area yang diisi ditentukan dengan parameter posisi x,y serta lebar dan tinggi. Untuk menentukan warna yang akan mengisi area maka fungsi *BSP_LCD_SetTextColor* harus dipanggil terlebih dahulu.

Listing Program 10.12 Fungsi BSP_LCD_FillRect

```
void BSP_LCD_FillRect(uint16_t Xpos, uint16_t Ypos, uint16_t Width,
                      uint16_t Height){
    BSP_LCD_SetTextColor(DrawProp.TextColor);
    do
    {
        BSP_LCD_DrawHLine(Xpos, Ypos++, Width);
    }
    while(Height--);

}
```

Sedangkan untuk menampilkan tulisan atau teks, maka setiap karakter dari teks tersebut harus digambar ke layar LCD, mengingat driver LCD TFT tidak mempunyai generator karakter seperti yang dimiliki oleh LCD karakter. Untuk bisa menampilkan karakter maka harus menggambar piksel per piksel sehingga terbentuk karakter yang diinginkan. Cara termudah adalah dengan menggunakan tabel data, ketika ada fungsi LCD untuk menampilkan sebuah kata atau sebuah karakter, maka kata tersebut akan dibaca per karakter, kemudian program akan membaca

tabel data sesuai dengan karakter yang terbaca dan menggambar karakter ke LCD berdasarkan tabel data tersebut.

Dalam contoh program di buku ini telah disediakan 5 contoh karakter dengan 5 macam font (jenis huruf) yang berbeda. Ketiga font mempunyai tabel data masing-masing yang disimpan di 5 file terpisah: font8.c, font12.c, font16.c, font20.c dan font24.c. Ke-5 font tersebut dideklarasikan sebagai variabel *Font8*, *Font12*, *Font16*, *Font20* dan *Font24* dengan tipe variabel sFONT. Tipe data sFONT ini mempunyai struktur data: pointer ke tabel data untuk menggambar font, lebar dan tinggi font dalam piksel.

Listing Program 10.13 Struktur Data sFONT

```
typedef struct _tFont
{
    const uint8_t *table;
    uint16_t Width;
    uint16_t Height;

} sFONT;
```

Sebagai contoh *Font24* mempunyai mempunyai tinggi 24 piksel dan lebar 17 piksel dengan tabel data *Font24_Table*.

Listing Program 10.14 Deklarasi Font24

```
sFONT Font8 = {
    Font24_Table,
    17, /* Width */
    24, /* Height */
};
```

Setiap karakter digambar di area sesuai dengan ukuran font, dalam hal ini 17x24 piksel. Gambar di bawah menunjukkan bagaimana huruf 'A' digambarkan di area 17x24 piksel. Di sebelahnya adalah data baris. Sebagai contoh baris pertama tidak ada piksel yang aktif, sehingga data untuk baris pertama adalah 0x00, 0x00, dan 0x00. Sedangkan di baris ke-4 ada beberapa piksel yang aktif sehingga datanya menjadi 0x1F, 0x80 dan 0x00. Piksel kolom pertama menjadi bit MSB dan seterusnya. Dan setiap karakter akan membutuhkan 72 byte (3 x 24).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1																	
2																	
3																	
4																	
5																	
6																	
7																	
8																	
9																	
10																	
11																	
12																	
13																	
14																	
15																	
16																	
17																	
18																	
19																	
20																	
21																	
22																	
23																	
24																	

Gambar 10.19 Contoh Font24 untuk Huruf A

Untuk menampilkan karakter telah disediakan fungsi *DrawChar()*. Fungsi ini pada dasarnya akan mengambil data-data dari tabel data yang telah ditentukan oleh font yang digunakan. Untuk mengambil data dari karakter yang akan ditampilkan diperlukan sebuah offset dari tabel data. Dalam perhitungan variabel *offset* ini didapat dari lebar font yang digunakan. Oleh karena data tabel disimpan dalam bentuk byte (8 bit), maka dalam perhitungan offset, lebar font (*width*) selalu ditambah dengan 7, untuk memastikan ketika dibagi 8 hasilnya tidak 0.

Listing Program 10.15 Fungsi untuk Menampilkan Karakter

```
static void DrawChar(uint16_t Xpos, uint16_t Ypos, const uint8_t *c){
    uint32_t i = 0, j = 0;
    uint16_t height, width;
    uint8_t offset;
    uint8_t *pchar;
    uint32_t line;

    height = DrawProp.pFont->Height;
    width = DrawProp.pFont->Width;

    offset = 8 * ((width + 7)/8) - width ;

    for(i = 0; i < height; i++)
    {
        pchar = ((uint8_t *)c + (width + 7)/8 * i);

        switch(((width + 7)/8))
```

```

{
    case 1:
        line = pchar[0];
        break;

    case 2:
        line = (pchar[0]<< 8) | pchar[1];
        break;

    case 3:
    default:
        line = (pchar[0]<< 16) | (pchar[1]<< 8) | pchar[2];
        break;
}

for (j = 0; j < width; j++)
{
    if(line & (1 << (width- j + offset- 1)))
    {
        BSP_LCD_DrawPixel((Xpos + j), Ypos, DrawProp.TextColor);
    }
    else
    {
        BSP_LCD_DrawPixel((Xpos + j), Ypos, DrawProp.BackColor);
    }
    Ypos++;
}
}

```

Parameter *c* dari fungsi *DrawChar* bukanlah karakter yang akan ditampilkan, melainkan sudah *pointer* kepada data dari karakter tersebut di dalam tabel font. Fungsi *BSP_LCD_DisplayChar*, yang memanggil fungsi *DrawChar*, akan menghitung pointer tersebut dari karakter yang akan ditampilkan.

Listing Program 10.16 Fungsi *BSP_LCD_DisplayChar*

```

void BSP_LCD_DisplayChar(uint16_t Xpos, uint16_t Ypos, uint8_t
Ascii){
    DrawChar(Xpos, Ypos, &DrawProp.pFont->table[(Ascii-' ') *
    DrawProp.pFont->Height * ((DrawProp.pFont->Width + 7) / 8)]);
}

```

Fungsi-fungsi lain yang terdapat di file *lcd.c*, misalnya fungsi untuk menggambar garis, horisontal dan vertikal, menggambar lingkaran, kotak, poligon, elips dan sebagainya.

10.4.3 MENAMPILKAN FILE BMP

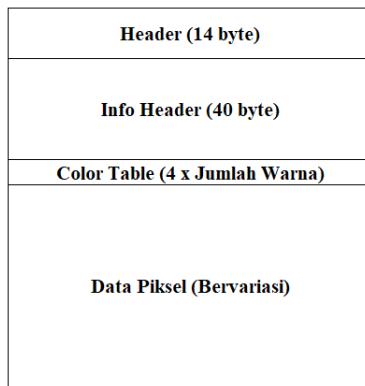
Dalam dunia digital (komputer), sebuah foto atau gambar disimpan dalam sebuah file dengan format tertentu. Dengan mengetahui format tersebut, mikrokontroler juga bisa diprogram untuk men-dekode format tersebut kemudian menampilkannya ke LCD. Tentu saja harus memperhatikan ukuran atau resolusi dari gambar tersebut, mengingat dengan keterbatasannya mikrokontroler hanya bisa menangani LCD dengan resolusi tinggi, sementara file gambar pada umumnya mempunyai resolusi tinggi. Banyak format yang digunakan untuk menyimpan gambar, di antaranya adalah format *bmp* dan *jpg* (atau *jpeg*). Kedua format file ini yang akan dibahas dibuku ini, format file *bmp* akan dibahas di sub-bab ini, sementara file *jpg* akan dibahas di sub-bab berikutnya.

10.4.3.1 Struktur File bmp

File *bmp* atau file *bitmap* atau disebut juga file *device independent bitmap* (*DIB*) merupakan format file yang dikembangkan oleh Microsoft untuk menyimpan gambar 2 dimensi, yang tidak tergantung pada perangkat display dan dikembangkan untuk sistem operasi Microsoft Windows dan OS/2. File *bitmap* bisa menyimpan gambar dengan warna hitam putih (*monokrom*) maupun berwarna dengan berbagai kedalaman warna (*color depth*). Kedalaman warna bisa 16 warna, 256 warna, 65 ribu warna (16 bit) atau 16 juta warna (24 bit). File *bmp* umumnya tidak dimampatkan (*compress*), walaupun format filenya mendukung pemampatan dengan algoritma RLE (*Run Length Encoding*) 4 atau 8 bit dan juga *Huffman 1D*..

File *bitmap* terdiri atas 4 bagian yaitu

1. *Header*.
2. *Info header*.
3. Tabel warna
4. Data gambar/piksel itu sendiri.



Gambar 10.20 Struktur File Bitmap

Header file bitmap berukuran 14 byte yang menyimpan informasi tentang tanda (*signature*) dari file bitmap (2 byte), ukuran file (4 byte), 4 byte disediakan mungkin pengembangan di masa depan, dan 4 byte yang berisi offset alamat dari awal file sampai alamat dari data gambar. Tanda dari file bitmap adalah 2 karakter "BM", 0x42 dan 0x4D, yang selalu ada di awal file bitmap. Empat byte berikutnya menyatakan ukuran dari file bitmap. Ukuran byte merupakan ukuran gambar itu sendiri dan ukuran header dan info header. Byte selanjutnya, dengan 4 byte, dinyatakan sebagai byte *reserved*, artinya untuk versi bitmap saat ini tidak menyatakan apa pun. Mungkin untuk pengembangan pada versi file bitmap selanjutnya. Bagian terakhir dari header adalah posisi byte dari data gambar dihitung dari byte awal file bitmap, ukurannya tentu saja tergantung dari ukuran/resolusi dari gambar itu sendiri.

Tabel 10.1 Struktur Header

Header	Ukuran	Offset	Keterangan
<i>Signature</i>	2 byte	0x0000	Karater "BM" atau 0x42, 0x4D
Ukuran File (<i>FileSize</i>)	4 byte	0x0002	Ukuran file dalam byte
<i>Reserved</i>	4 byte	0x0006	Tidak digunakan, selalu bernilai 0
Offset Data (<i>DataOffset</i>)	4 byte	0x000A	Offset dari awal file ke awal data bitmap

Bagian selanjutnya dari struktur file bitmap adalah *InfoHeader*, bagian ini mempunyai ukuran 40 byte dan menyimpan informasi mengenai bitmap yang tersimpan dalam file. InfoHeader akan menyimpan ukuran atau

resolusi gambar (lebar dan tinggi), jumlah bit per piksel, jenis kompresi (walaupun file bitmap biasanya tidak dikompres) dan informasi-informasi lain seperti ditunjukkan oleh tabel.

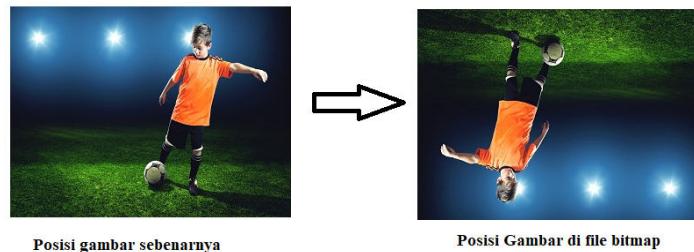
Tabel 10.2 Struktur InfoHeader

InfoHeader	Ukuran	Offset	Keterangan
Ukuran (Size)	4 byte	0x000E	Ukuran dari InfoHeader, selalu bernilai 40 (0x00000028)
Lebar (Width)	4 byte	0x0012	Lebar (ukuran horisontal) dalam piksel
Tinggi (Height)	4 byte	0x0006	Tidak digunakan, selalu bernilai 0
Planes	2 byte	0x001A	Jumlah <i>plane</i> warna (selalu bernilai 1)
Bit per piksel	2 byte	0x001C	Bit per piksel digunakan untuk menyimpan informasi jumlah warna dari file bitmap. Kemungkinan nilainya: <ul style="list-style-type: none"> • 1, untuk gambar monokrom, jumlah warna 1. • 4 = 4 bit warna (16 warna) • 8 = 8 bit warna (256 warna) • 16 = 16 bit warna (65536 warna) • 24 = 24 bit warna (16 juta warna) • 32 = 32 bit warna (4 miliar warna)
Kompresi (Compression)	4 byte	0x001E	Jenis kompresi <ul style="list-style-type: none"> • 0 = BI_RGB (tidak ada kompresi) • 1 = BI_RLE8 (enkode 8 bit RLE) • 2 = BI_RLE4 (enkode 4 bit RLE)
Ukuran gambar (ImageSize)	4 byte	0x0022	Ukuran data gambar. Nilai 0 bisa diberikan untuk file dengan jenis kompresi 0 (BI_RGB)
X piksel per meter (XpixelsPerM)	4 byte	0x0026	Resolusi horisontal dalam piksel/meter
Y piksel per meter (YpixelsPerM)	4 byte	0x002A	Resolusi vertikal dalam piksel/meter
Jumlah warna yang digunakan	4 byte	0x002E	Jumlah warna yang digunakan, sebagai contoh untuk warna 8 bit akan bernilai 256 (0x100)
Warana-warna penting	4 byte	0x0032	Jumlah warna-warna penting, akan bernilai 0 jika semua warna adalah penting

Tabel warna (*color table*), bagian selanjutnya dari struktur file bitmap, biasanya berukuran 4 byte (offset 0x0036), diperlukan untuk gambar dengan warna kurang dari 8 bit. Tiga byte pertama adalah intensitas

untuk warna merah, hijau dan biru, sedangkan byte terakhir tidak dipakai.

Bagian terakhir dari file bitmap adalah data gambar itu sendiri. Ukuran dari bagian ini ditentukan oleh resolusi atau ukuran gambar tersebut dan ukuran bit per piksel dari gambar, seperti ditunjukkan oleh bagian *ImageSize* dari *InfoHeader*.



Gambar 10.21 *Posisi Gambar di File Bitmap*

The screenshot shows a Windows Notepad window titled "image_02.txt - Notepad". The window displays a hex dump of a bitmap file. The title bar includes the file name, application name ("Notepad"), and version information ("Turbo Dump Version 6.5.4.0 Copyright (c) 1988-2016 Embarcadero Technologies, Inc."). The menu bar includes File, Edit, Format, View, Help. The main content area shows the following hex dump:

```
000000: 42 4D 36 84 03 00 00 00 00 00 36 00 00 00 28 00 BM6.....6...(.  
000010: 00 00 40 01 00 00 F0 00 00 00 01 00 18 00 00 ..@.....  
000020: 00 00 00 84 03 00 23 2E 00 00 23 2E 00 00 00 00 .....#...#....  
000030: 00 00 00 00 00 14 A9 A5 1D B2 AE 1E B1 AD 15 .....  
000040: AA A6 00 8F 8A 2C CA C4 00 9E 98 14 B9 B2 00 A7 .....  
000050: 9F 00 AD A4 06 B1 A9 18 C0 B9 13 BE B6 1F C7 C0 .....  
000060: 1D C2 BB 0E B3 AB 14 BB B2 09 B1 A5 0A A8 A1 0E .....  
000070: AF A7 0F B5 AE 23 C9 C2 08 A8 A2 00 92 8C 00 A4 .....#....  
000080: 9B 0B B4 AB 07 A8 9E 21 C5 BA 09 B7 AD 00 AB A1 .....!  
000090: 2A D2 CB 0B B4 AB 06 B5 AB 00 AF A2 0E B7 AE 09 *.....  
0000A0: AE A6 16 B8 B3 00 8F 8A 0E BC B5 16 C8 C1 25 D0 .....%.  
0000B0: C8 1A BA B4 00 81 7D 21 BB B6 17 D4 CB 17 E0 D5 .....}!.....  
0000C0: 0C C8 C2 00 B5 AD 18 CD C4 17 B8 B0 5E D3 D0 53 .....^.S  
0000D0: C8 C5 0E AD A3 00 AC 9F 0E B5 AC 1D C4 BB 13 BD .....  
0000E0: B7 03 B3 AC 01 B7 AF 29 D1 CB 38 BA B9 3F C1 C0 .....).8.?..  
0000F0: 29 CA C2 07 B5 AD 24 D0 CC 23 C9 C8 1C C1 BE 32 ).....$..#....2  
000100: D3 D1 26 BF BC 15 B4 B0 01 AE A6 1F D5 CA 11 C3 ..&.....  
000110: B8 1A C9 BC 35 DC CF 28 C5 BA 44 C8 C1 5D D7 D1 ....5..+.D..].  
000120: 17 89 82 30 B3 A9 22 C6 BB 10 CB BC 1C DE D1 11 ...0..".  
000130: D6 C8 0A CA BF 08 C4 B9 0E C6 BE 26 DE D6 0B C3 .....&....  
000140: BD 04 BE B8 18 D0 CA 07 C1 BB 09 C8 C3 14 D6 D0 .....  
000150: 17 D9 D3 0B C7 C2 3C E8 E4 12 BE BA 0A BF B7 1D .....<.....  
000160: D5 CD 2A DF D7 0F C5 BD 0D CA C1 1B D8 CF 1A D0 ..*.....  
000170: C8 2B DB D4 06 AE A8 20 CA C4 11 C3 BC 1D DA D1 .+....  
000180: 0B D0 C8 17 D7 D0 33 DF D9 24 CE C8 09 C1 B9 16 .....3..$.  
000190: CC C4 38 D7 D3 20 BF BB 20 D6 CE 0F D0 C6 08 C7 .8.. . ....  
0001A0: C0 10 D0 C9 17 D7 D0 0C CE C7 18 D8 D1 2E EA E4 .....  
0001B0: 1E CD CA 2E DA D6 1A C8 C1 0A B8 B0 17 C7 C0 08 .....
```

Gambar 10.22 Isi Sebuah File Bitmap

by Kang U2Man (u_2man@yahoo.co.id)

by Kang U2Man (u_2man@yahoo.co.id)

Bab 11

BERMAIN-MAIN DENGAN DEKODER MP3 + MIDI

by Kang U-2Man (u_2man@yahoo.co.jp)

by Kang U2Man (u_2man@yahoo.co.id)

Bab 12

BERMAIN-MAIN DENGAN FILE VIDEO

by Kang U-2Man (u_2man@yahoo.co.id)

by Kang U2Man (u_2man@yahoo.co.id)

Bab 13

BERMAIN-MAIN DENGAN KAMERA DIGITAL

by Kang U-2Man (u_2man@yahoo.co.id)

by Kang U2Man (u_2man@yahoo.co.id)

Bab 14

BERMAIN-MAIN DENGAN USB

by Kang U2Man (u_2man@yahoo.co.id)

by Kang U2Man (u_2man@yahoo.co.id)

Bab 15

BERMAIN-MAIN DENGAN ETHERNET

by Kang U-2Man (u_2man@yahoo.co.id)

by Kang U2Man (u_2man@yahoo.co.id)

Bab 16

BERMAIN-MAIN DENGAN RTOS

by Kang U2Man (u_2man@yahoo.co.id)S

by Kang U2Man (u_2man@yahoo.co.id)

Bab 17

BERMAIN-MAIN DENGAN IAP

by Kang U2Man (u_2man@yahoo.co.id) JP

by Kang U2Man (u_2man@yahoo.co.id)

DAFTAR PUSTAKA

- , AMBA 3 AHB-Lite Protocol v1.0 Specification, ARM Limited, 2001
- , AMBA Specification (Rev 2.0), ARM Limited, 199
- , Application Note Atmel AT02346: Using the MPU on Atmel Cortex-M3/Cortex-M4 based Microcontroller, Atmel, 2013
- , Application Note Extending the DAC Performance of STM32 Microcontrollers, ST Microelectronics, 2015
- , Application Note General-purpose Timer Cookbook, ST Microelectronics, 2016
- , Application Note Getting Started with STM32F20xx/21xx MCU Hardware Development, ST Microelectronics, 2011
- , Application Note How to get the best ADC accuracy in STM32Fx Series and STM32L1 Series Devices, ST Microelectronics, 2013
- , Application Note LCD-TFT display controller (LTDC) on STM32 MCU, ST Microelectronics, 2008
- , Application Note Oscillator Design Guide for STM8AF/AL/S and STM32 Microcontrollers, ST Microelectronics, 2017
- , Application Note STM32 Cross-series Timer Overview, ST Microelectronics, 2016
- , Application Note STM32 Microcontroller System Memory Boot Mode, ST Microelectronics, 2016
- , Application Note STM32's ADC Modes and Their Applications, ST Microelectronics, 2010
- , Application note TFT LCD Interfacing with high-density STM32F10xxx FSMC, ST Microelectronics, 2008
- , Application Note Using Cortex-M3 and Cortex-M4 Fault Exceptions, Keil, 2010

- , *Application Note Using STM32 Device PWM Shut-down Features for Motor Control and Digital Power Conversion*, ST Microelectronics, 2016
 - , *ARM Accredited MCU Engineer AAME Syllabus*, ARM Limited, 2014
 - , *ARM Cortex-M3 Introduction*, ARM University Relations, 2010
 - , *ARMv7-M Architecture Reference Manual*, ARM Limited, 2010
 - , *Cortex-M3 Devices Generic User Guide*, ARM Limited, 2010
 - , *Cortex-M3 Instruction Set Technical User's Manual*, Texas Instruments, 2010
 - , *Cortex-M3 Technical Reference Manual Revision r2p1*, ARM Limited, 2008
 - , *Cortex-M3 Technical Reference Manual*, ARM Limited, 2008
 - , *Programming Manual STM32F10xxx/20xxx/21xxx/L1xxxx Cortex-M3 Programming Manual*, ST Microelectronics, 2013
 - , *Reference Manual STM32F205xx, STM32F207xx, STM32F215xx and STM32F217xx Advanced ARM-based 32-bit MCUs*, ST Microelectronics, 2015
 - , *SD Card Specification Simplified Version of: Part E1 SD Input/Output (SDIO) Card Specification Version 1.00*, SD Association, 2001
 - , *SD Specifications Part 1 Physical Layer Simplified Specifications Version 6.00*, Technical Committee SD Card Association, 2018
 - , *User Manual Developing Applications on STM32Cube with FatFS*, ST Microelectronics, 2014
- Joseph Yiu, *The Definitive Guide To ARM Cortex-M3 Processors Second Edition*, Newnes, 2007
- Richard Phelan, *Improving ARM Code Density and Performance New Thumb Extensions to the ARM Architecture*, ARM Limited, 2003
- Shyam Sadasivan, *White Paper An Introduction to the ARM Cortex-M3 Processor*, -, 2006
- Trevor Martin, *The Insider's Guide To The STM32 ARM Based Microcontroller An Engineer's Introduction To The STM32 Series Version 1.8*, www.hitex.com, 2009

by Kang U2Man (u_2man@yahoo.co.id)