Pintaros

Ricky Hariady

June 17, 2014

Chapter 1

Gambaran Umum

PintarOS merupakan sistem operasi untuk *smart card*. Desain sistem operasi akan mengikuti sebagian dari standard ISO 7816 yang mengatur berbagai aspek mengenai *smart card*.

pintarOS dirancang sebagai sebuah sistem operasi yang generic, yang memiliki tingkat keterbebasan yang tinggi terhadap hardware sehingga tidak terbatas pada arsitektur komputasi tertentu. Meskipun sebagai awal dikembangkan untuk smart card berbasis 8051, pintarOS dirancang agar dapat dengan mudah diimplementasikan pada arsitektur mikroprocessor lainnya ataupun platform tertentu.

PintarOR dirancang sebagai sistem operasi yang bersifat *platform independent* sehingga tidak terbatas pada platform hardware tertentu saja. Sebagai awal pintarOS akan diimplementasikan pada smart card berbasis mikroprosesor AT90S8515 dan EEPROM 24Cxx atau yang lebih sering disebut *funcard*, namun nantinya akan dapat diimplementasikan (porting) pada berbagai smart card dengan platform hardware lainnya menggunakan desain dan kode sumber yang sama dengan perubahan yang minimum. Sifat platform-independent ini dapat dicapai dengan memisahkan bagian sistem yang bergantung pada platform hardware dalam sebuah modul tersendiri dan yang disebut HAL (Hardware Abstraction Layer), yang menyediakan antarmuka yang sama untuk setiap platform hardware bagi bagian-bagian sistem lainnya.

1.0.1 Arsitektur Sistem

PintarOS dirancang sebagai sistem yang modular, dimana setiap module memiliki fungsi-fungsi yang berbeda. Gambar x menampilkan arsitektur yang digunakan oleh pintarOS, dimana terdiri dari beberapa lapisan. Di bagian paling dasar adalah lapisan Hardware yang menjadi platform dari

smart card sendiri, sementara di bagian paling atas adalah lapisan aplikasi. PintarOS berada diantara keduanya, yang memungkinkan aplikasi berjalan diatas platform smart card.

Sebagian dari sistem operasi pintarOS ini diletakkan pada memory ROM/Flash dari smart card : HAL Driver, transmission handler, general command handler, etc. sementara sebagian lainnya diletakkan pada EEPROM. Bagian yang diletakkan pada pada ROM/Flash merupakan kode-kode program dari setiap module-module. Sementara yang diletakkan pada EEPROM merupakan data-data yang akan digunakan oleh pintarOS dalam menjalankan fungsinya seperti Cryptography Key,

Sebagaimana telah disebutkan, pintarOS dirancang secara modular. Modul-modul utama dari pintarOS ini ditampilkan pada Gambar 1.1. Modul-modul ini dipisahkan menjadi dua bagian, yaitu hardware-dependent dan hardware-independent. bagian hardware-dependent terutama terdiri dari modul-modul hardware abstraction layer (HAL). Modul-Modul ini berfungsi seperti driver yang mengabstraksi hardware pada software sehingga fungsi-fungsi hardware dapat digunakan dengan cara yang sama sebagai sebuah layanan oleh modul lainnya meskipun menggunakan platform hardware yang berbeda.

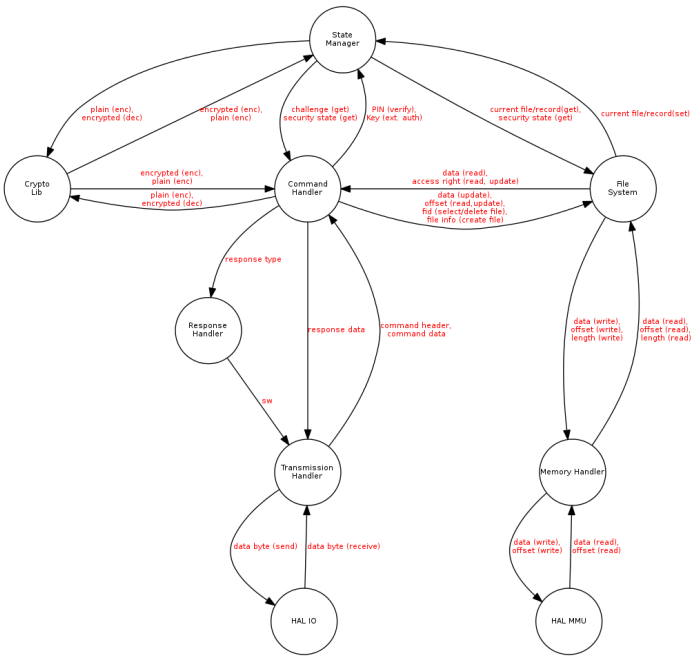

Figure 1.1: DFD Keseluruhan Sistem

Berikut adalah penjelasan fungsi dari setiap modul:

Hardware Abstraction Layer (HAL)

berfungsi menyediakan abstraksi dari hardware yang ada dan menyediakan layanan yang sama pada module-module di lapisan yang lebih tinggi meskipun menggunakan platform hardware yang berbeda.

Transmission Handler
bertanggung jawab menangani protokol transmisi yang dipilih untuk berkomunikasi dengan terminal.

Memory Handler
bertanggung jawab menangani pembacaan dan penulisan data secara umum dan generic pada media penyimpanan yang ada

Command Handler
bertanggung jawab menginterpretasikan APDU Commmand dan memanggil command handler yang sesuai,yang kemudian akan melaksanakan instruksi sebagaimana yang diminta oleh command APDU. Setiap instruksi dijalankan menggunakan pendekatan yang sama berdasarkan pada context command APDU. Setiap instruksi yang didukung harus memiliki command handlernya masing-masing.

Response Manager
berfungsi membentuk pesan response APDU (SW1 SW2) berdasarkan response type dari command handler.

File System
berfungsi menangani segala hal mengenai file.

State Manager
berfungsi menyimpan state smart card. Beberapa modul akan menggunakan state ini dalam menjalankan fungsinya, seperti File System membutuhkan informasi mengenai file yang sedang dipilih ketika akan membaca sebuah file.

Crypto Lib
menyediakan layanan kriptografi seperti enkripsi dan dekripsi.

1.0.2 File System

Bagian ini menjelaskan secara umum mengenai *file system* yang akan digunakan pada *pintarOS*. Sebagaimana dapat dilihat pada rancangan utama, *file system* berfungsi menangani struktur file dan operasi-operasi yang dapat dilakukan terhadap file tersebut. Bagaimana data file tersebut disimpan pada memory fisik tidak menjadi tanggung jawab File System karena akan

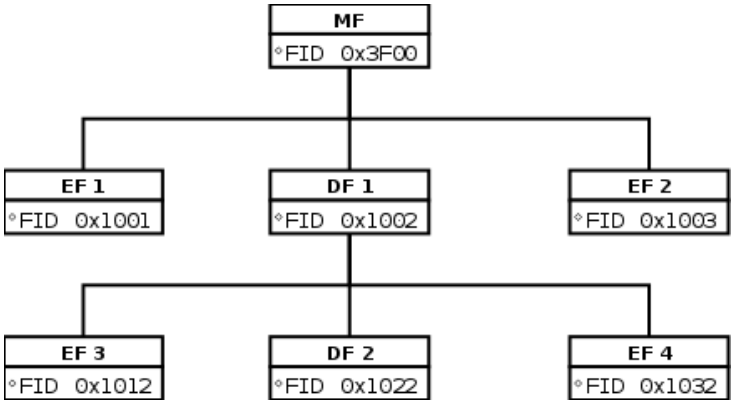

Figure 1.2: struktur file system

dilakukan oleh *HAL Memory* dengan menyediakan layanan untuk penyimpanan dan pengambilan data dari Non-Volatile Memory (NVM) menggunakan pengalamatan yang linier dan kontinyu.

Struktur File System

Sebagaimana dijelaskan pada dokumen ISO 7816-4, terdapat dua jenis file dasar yang dapat digunakan pada *smart card* yaitu :

- Elementary File (EF), dan
- Dedicated File (DF).

Elementary File merupakan jenis file yang dapat menyimpan data aktual. Sedangkan *dedicated File* merupakan file khusus yang tidak berfungsi menyimpan data aktual, melainkan memuat daftar sejumlah file lainnya (dapat berupa EF maupun DF). Fungsi DF ini dapat disamakan dengan fungsi *directory* pada komputer PC. Dengan mengelompokkan sejumlah file yang saling berkaitan dalam sebuah DF (sebagai contoh dalam sebuah aplikasi) akan menghasilkan sebuah sistem file yang terstruktur. Sejumlah file DF ini kemudian dapat dikelompokkan kembali dalam file DF lainnya dengan level yang lebih tinggi. Sebuah *Dedicated File* khusus yang menempati lokasi tertinggi (*root*) dalam struktur file system kemudian disebut sebagai *Master File* (MF). Gambar 1.2 menampilkan contoh struktur file system dari sebuah *smart card* yang terdiri dari sebuah MF serta sejumlah DF dan EF.

Atribut File

Setiap file pada *smart card* dikenali melalui FID dari file tersebut. Untuk mencegah ambiguitas, maka file-file yang berada dalam 1 DF tidak boleh

memiliki FID yang sama (demikian pula dengan DF induknya).

Dilihat dari sisi penggunanya, *Elementary File* (EF) dibagi menjadi dua jenis, yaitu :

- Working EF, dan
- Internal EF.

Working EF adalah EF yang menyimpan data-data milik aplikasi *smart card* dan dapat diakses oleh aplikasi. Sementara *internal* EF adalah EF yang menyimpan data-data milik *operating system* dan tidak dapat diakses langsung oleh aplikasi. Contoh *internal* EF misalnya adalah file EF yang menyimpan kunci keamanan. Kunci yang disimpan dalam file ini akan dibaca oleh *operating syste*m dan dicocokkan dengan kunci yang diberikan oleh pengguna *smart card* untuk memperoleh hak untuk mengakses file-file lainnya didalam DF.

Standar ISO 7816-4 memberikan 5 struktur internal berbeda yang mungkin digunakan pada sebuah file EF, yaitu :

- Transparent
- Linier Record (fixed length)
- Linier Record (variable length)
- Cyclic Record
- TLV

Selain jenis (*working* atau *internal*) dan struktur internalnya EF, terdapat sejumlah atribut lainnya yang dapat melekat pada sebuah file, diantaranya :

- read-write/read-only
- shareable/not-shareable
- access condition

Operasi file

Sebagaimana pada sistem komputer lainnya, pengguna dapat melakukan sejumlah operasi terhadap file. Operasi-operasi file yang dapat dilakukan pada *smart card* mencakup:

- Menciptakan file

- Menghapus file
- Memilih file
- Menulis file
- Membaca file
- Mengubah atribut file

Keamanan file

Salah satu faktor penting yang menjadi kelebihan *smart card* dibanding kartu elektronik lainnya adalah dari segi keamanan. Demikian pula terhadap file yang disimpan didalamnya (dalam hal ini oleh *file system*). Untuk mencapai tingkat keamanan tersebut, pengguna harus terlebih dahulu memenuhi kondisi tertentu untuk dapat mengakses sebuah file. Kondisi yang diperlukan ini selanjutnya disebut juga sebagai *Access Condition* (AC). Metode yang akan digunakan untuk mengimplementasikan *access condition* ini adalah dengan menggunakan *security state*. Pada metode ini, pengguna harus mencapai *security state* tertentu (melalui verifikasi PIN) untuk dapat mengakses sebuah file. *Security state* yang dibutuhkan dapat berbeda untuk setiap operasi, sehingga dibutuhkan sebuah access condition untuk setiap operasi.

- access condition untuk membaca,
- access condition untuk menulis,
- access condition untuk menghapus file,
- access condition untuk menciptakan file baru (khusus DF).
- access condition untuk mengubah atribut

Chapter 2

Hardware Abstraction Layer

Hardware Abstraction Layer (HAL) berfungsi mengabstraksi hardware dan menyediakan layanan-layanan yang sama terlepas dari paltform hardware yang digunakan. HAL dapat dianalogikan seperti *device driver* pada sistem komputer PC karena akan berhubungan langsung dengan hardware. Tabel 2.1 menampilkan antarmuka yang disediakan HAL untuk digunakan oleh modul-modul lainnya.

Nama Fungsi	**Kegunaan**
HW Init	Menginisialisasi Hardware
IO Receive	Menerima 1 byte data melalui port IO
IO Transmit	Mengirimkan 1 byte data melalui port IO
Memory Internal Read	Membaca 1 byte data dari internal memory
Memory Internal Write	Menulis 1 byte data dari internal memory
Memory Eksternal Read	Membaca 1 byte data dari external memory
Memory Eksternal Write	Menulis 1 byte data dari external memory

Table 2.1: Daftar antarmuka fungsi yang disediakan HAL

2.1 HW Init

Berfungsi mempersiapkan HW untuk menyediakan layanan bagi modul-modul lainnya. Gambar 2.1 menampilkan DFD dari fungsi HW Init. Implementasi dari Fungsi ini sangat bergantung pada platform smartcard yang akan digunakan.

Gambar 2.2 menampilkan diagram alir fungsi HW Init pada smartcard berbasis funcard (uC AT90S8515 EEPROM 24C64).

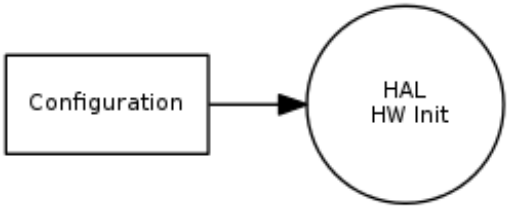

Figure 2.1: DFD HW Init

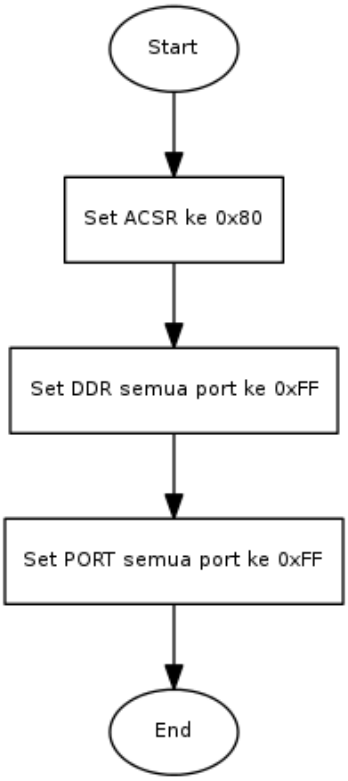

Figure 2.2: Flowchart HW Init

Pertama fungsi hw init akan menginisialisasi pin ke 6 dari port B dari mikrokontroler sebagai port Output dengan men-set Data Direction Register (DDR) 0x40, dan menginisialisasi nilainya menjadi 1 (high) dengan men-set Data Register (PORTB) menjadi 0x40.

Selanjutnya fungsi ini akan mematikan modul Analog Comparator pada mikrokontroler dengan tujuan mengurangi konsumsi daya dengan mengeset bit ke-7 (ACD - Analog Comparator Disable) dari ACSR (Analog Comparator Control and Status Register).

2.1.1 Pengujian

Output
DDRB = 0x40
PORTB = 0x40
ACSR = 0x80

Table 2.2: Test Vector Fungsi HAL HW Init

Tabel 2.2 menampilkan Test Vector yang digunakan untuk menguji fungsi HAL HW Init.

2.1.2 Implementasi

Tabel 2.3 menampilkan purwarupa dari implementasi fungsi HAL HW Init.

Name	HAL_HWInit
Input	-
Output	-

Table 2.3: Prototype Fungsi HW Init

Listing 2.1 menampilkan potongan program yang mengimplementasi fungsi HAL HW Init.

```
1 void HAL_HWInit()
2 {
3    outb(DDRB ,0x40);
4    outb(PORTB,0x40);
5    outb(ACSR ,0x80);
6 }
```

Listing 2.1: Listing Program Fungsi HAL HW Init

2.2 IO Receive Byte T0

Berfungsi menerima 1 byte data dari Port IO menggunakan protokol komunikasi T0. Gambar 2.3 menampilkan DFD dari fungsi HAL IO Receive Byte T0(RxByte). Implementasi dari Fungsi ini sangat bergantung pada platform

smartcard dimana pintarOS akan digunakan. Gambar 2.4 menampilkan diagram alir dari fungsi Receive Byte T0 untuk *smart card* funcard.

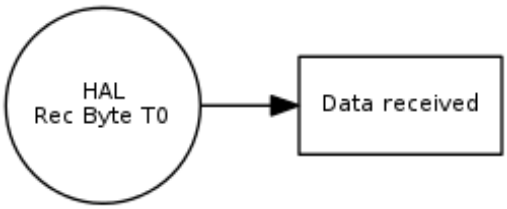

Figure 2.3: DFD Receive Byte T0

2.2.1 Pengujian

Input	**Output**
Databyte (from terminal)	**Return Value**
0x00	= 0x00
0x01	= 0x01
...	
...	
0xff	= 0xff

Table 2.4: Test Vector Fungsi HAL Receive Byte T0

Tabel 3.2 menampilkan Test Vector yang digunakan untuk menguji fungsi HAL Receive Byte T0.

2.2.2 Implementasi

Tabel 2.5 menampilkan purwarupa dari implementasi fungsi HAL IO Receive Byte T0.

Name	HAL_IO_RxByteT0
Input	-
Output	data yang diterima (1 byte)

Table 2.5: Prototype Fungsi HAL IO Receive Byte T0

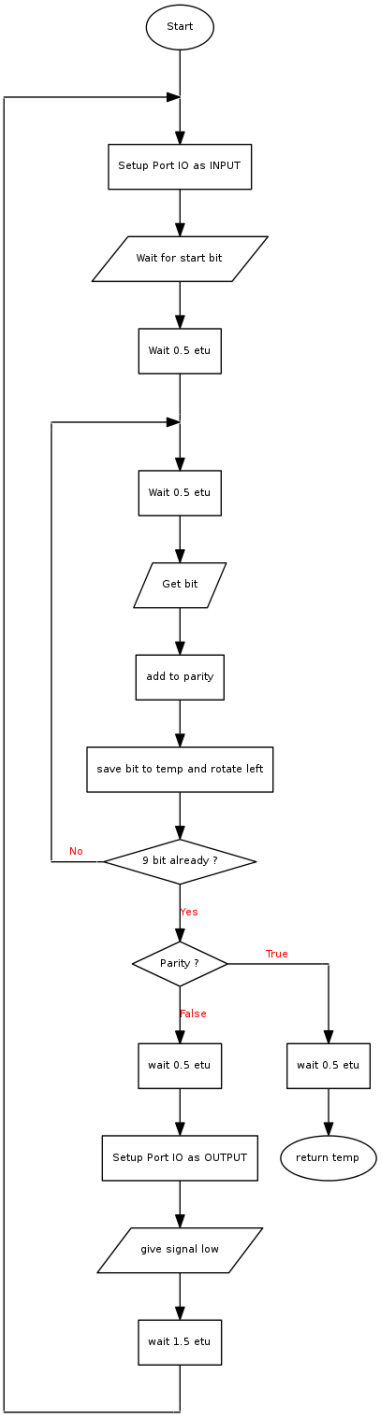

Figure 2.4: Flowchart Receive Byte T0

Listing 2.2 menampilkan potongan program yang mengimplementasi fungsi HAL IO Receive Byte.

```
;=============================================================================
; Receive a byte with T=0 error correction.
; result r25(=0):r24
HAL_IO_RxByteT0:
   push r23           ; 2 - getbit
   push r22           ; 2 - delay
   push r21           ; 2 - loop counter
   push r20           ; 2 - parity counter

   ; Set direction bit, to indicate, that we received a byte
   ldi     r22, 1
   sts     direction,r22

restartrecbyte:
   ; Setup IN direction
   cbi     DDRB, 6       ; 2
   cbi     PORTB, 6   ; 2

; Wait for start bit.
waitforstart:
   ; Bit begins here.
   sbic PINB, IO_PIN  ; 1/2!
   rjmp waitforstart  ; 2/0
   sbic PINB, IO_PIN  ; 1/2! - Recheck for spike
   rjmp waitforstart  ; 2/0
   ; Sample start bit
   clr     r24           ; 1
   clr     r25           ; 1 - Clear zero byte for ADC
   ldi     r22, 31       ; 1
   rcall delay     ; 100
   rcall getbit       ; 3 (16bit PC)
   ;brcs waitforstart ; 1/2 - Go on, even if not valid a start bit?
   nop                ; 1 - For brcs
; Receive now 9 bits
   ldi     r21, 0x09     ; 1
   clr     r20           ; 1
   ldi     r22, 66       ; 1
   nop                ; 1
   nop                ; 1
rnextbit:
```

```
41    rcall delay      ; 205/202
42    rcall getbit       ; 3
43    add     r20, r23   ; 1
44    clc                ; 1
45    sbrc r23, 0        ; 1/2
46    sec                ; 1/0
47    ror     r24          ; 1
48    ldi     r22, 65      ; 1
49    dec     r21          ; 1
50    brne rnextbit   ; 1/2
51 ; Check parity
52    rol     r24          ; 1 - We've rotated one to much
53    sbrc r20, 0        ; 1/2
54    rjmp regetbyte     ; 2/0
55
56    ; Wait halve etu
57    ldi     r22, 76      ; 1
58    rcall delay      ; 235 - Precise enough
59
60    clr     r25
61    pop     r20          ; 2 - parity counter
62    pop     r21          ; 2 - loop counter
63    pop     r22          ; 2 - delay
64    pop     r23          ; 2 - getbit
65    ret
66
67 regetbyte:
68    ; Wait halve etu
69    ldi     r22, 76      ; 1
70    rcall delay      ; 235 - Precise enough
71    ; Set OUT direction
72    sbi     DDRB, 6      ; 2
73    ; Signal low
74    cbi     PORTB, 6   ; 2
75    ldi     r22, 182   ; 2
76    rcall delay      ; 553 - about 1.5 etu
77    rjmp restartrecbyte ; 2
78
79 ;=============================================================================
80 ; Read a bit.
81 ; Uses r23, r25
82 ; Returns bit in r23.0.
83 ; 5 cycles before first bit
```

```
84  ; 8 cycles after last bit.
85  getbit:
86     clr     r23           ; 1
87     clc                   ; 1
88     ; At start + 112 cycles
89     sbic  PINB, IO_PIN  ; 1/2
90     sec                   ; 1/0
91     adc     r23, r25    ; 1
92     rcall intrabitdelay ; 70
93     clc                   ; 1
94     ; At start + 186 cycles
95     sbic  PINB, IO_PIN  ; 1/2
96     sec                   ; 1/0
97     adc     r23, r25    ; 1
98     rcall intrabitdelay ; 70
99     clc                   ; 1
100    ; At start + 260 cycles
101    sbic  PINB, IO_PIN  ; 1/2
102    sec                   ; 1/0
103    adc     r23, r25    ; 1
104    ; Get second bit of the sum.
105    lsr     r23           ; 1
106    ret                   ; 4   (with 16bit PC)
```

Listing 2.2: Listing Program Fungsi HAL IO Receive Byte T0

2.3 IO Transmit Byte T0

Berfungsi mengirimkan 1 byte data melalui Port IO menggunakan protokol komunikasi T0. Gambar 2.5 menampilkan DFD dari fungsi HAL IO Transmit Byte T0(RxByte). Gambar 2.6 menampilkan diagram alir dari fungsi Transmit Byte T0 untuk *smart card* funcard.

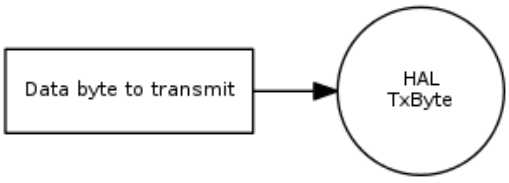

Figure 2.5: DFD Transmit Byte T0

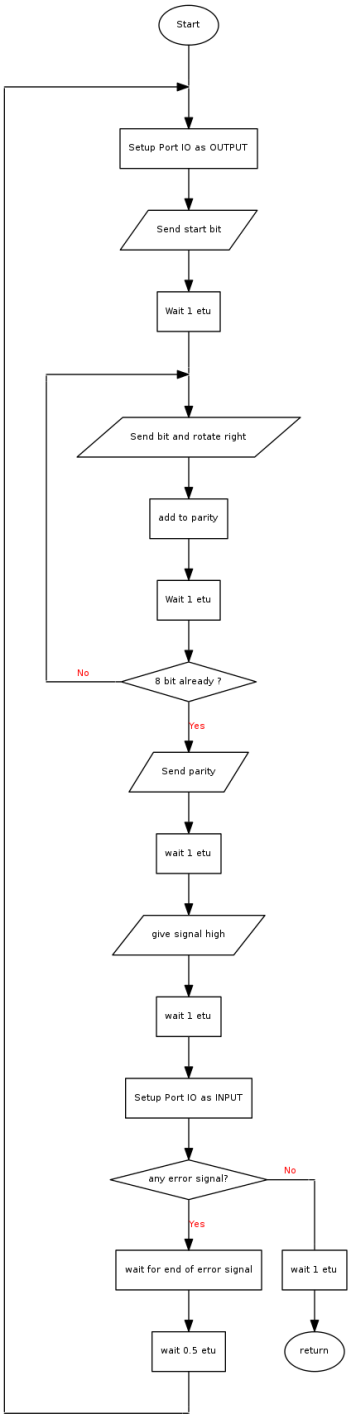

Figure 2.6: Flowchart HAL IO Transmit Byte T0

2.3.1 Pengujian

Input	Output
Data	**Databyte (to terminal)**
0x00	= 0x00
0x01	= 0x01
...	
...	
0xff	= 0xff

Table 2.6: Test Vector Fungsi HAL IO Transmit Byte T0

Tabel 2.6 menampilkan Test Vector yang digunakan untuk menguji fungsi HAL IO Transmit Byte T0.

2.3.2 Implementasi

Tabel 2.7 menampilkan purwarupa dari implementasi fungsi HAL IO Transmit Byte T0.

Name	HAL_TxByteT0
Input	data yang akan dikirimkan (1 byte)
Output	-

Table 2.7: Prototype Fungsi HAL IO Transmit Byte T0

Listing 2.3 menampilkan potongan program yang mengimplementasi fungsi HAL IO Transmit Byte T0.

```
;=====================================================================
; Send a byte with T=0 error correction.
; byte r25(=0):r24
HAL\_TxByteT0:
   push r22           ; 2 - delay
   push r23           ; 2 - parity counter

   lds     r22,direction
   tst     r22
   breq resendbytet0
```

```
11    rcall delay1etu    ;
12    rcall delay1etu    ;
13    ; Clear direction bit, to indicate, that we sent a byte
14    ldi     r22, 0
15    sts     direction,r22
16
17 resendbytet0:
18    ; Set OUT direction
19    sbi     PORTB, 6   ; 2
20    sbi     DDRB, 6       ; 2
21    ; Send start bit
22    cbi     PORTB, IO_PIN ; 2
23    ldi     r22, 119   ; 1
24    rcall delay      ; 364
25    ; Send now 8 bits
26    ldi     r25, 0x08     ; 1
27    clr     r23           ; 1
28 snextbit:
29    ror     r24           ; 1
30    brcs  sendbit1   ; 1/2
31    cbi     PORTB, IO_PIN ; 2
32    rjmp  bitset        ; 2
33 sendbit1:
34    sbi     PORTB, IO_PIN ; 2
35    inc     r23           ; 1
36 bitset:
37    ldi     r22, 118   ; 1
38    rcall delay      ; 361
39    nop                ; 1
40    dec     r25           ; 1
41    brne  snextbit   ; 1/2
42    ; Send parity
43    sbrc  r23, 0        ; 1/2
44    rjmp  sendparity1   ; 2
45    nop                ; 1
46    nop                ; 1
47    cbi     PORTB, IO_PIN ; 2
48    rjmp  delayparity   ; 2
49 sendparity1:
50    nop                ; 1
51    sbi     PORTB, IO_PIN ; 2
52    nop                ; 1
53    nop                ; 1
```

```
54 delayparity:
55    ldi     r22, 112   ; 1
56    rcall delay      ; 343
57    ; Stop bit
58    sbi     PORTB, IO_PIN ; 2
59    ldi     r22, 119   ; 1
60    rcall delay      ; 364
61    ; Set IN direction
62    cbi     DDRB, 6       ; 2
63    cbi     PORTB, 6   ; 2
64    ; Look for error signal
65    clc                ; 1
66    sbic PINB, IO_PIN  ; 1/2
67    sec                ; 1/0
68    brcs retsendbytet0 ; 1/2
69    ; Resend byte
70    ; Bring byte to starting position
71    ror     r24           ; 1
72    ; Wait for end of error signal
73 waitforendoferror:
74    sbic PINB, IO_PIN  ; 1/2!
75    rjmp waitforendoferror ; 2/0
76    ; Wait then a halve etu
77    ldi     r22, 58       ; 1
78    rcall delay      ; 181
79    rjmp resendbytet0  ; 2
80    ; return
81 retsendbytet0:
82    ldi     r22, 116   ; 1
83    rcall delay      ; 355
84    pop     r23           ; 2 - parity counter
85    pop     r22           ; 2 - delay
86    ret                ; 4
87 ;=============================================================================
```

Listing 2.3: Listing Program Fungsi HAL IO Transmit Byte T0

2.4 Internal Memory Read Byte

Berfungsi membaca 1 byte data dari Memory EEPROM Internal. Gambar 2.7 menampilkan DFD dari fungsi HAL Internal Memory Read Byte (Read-Byte).

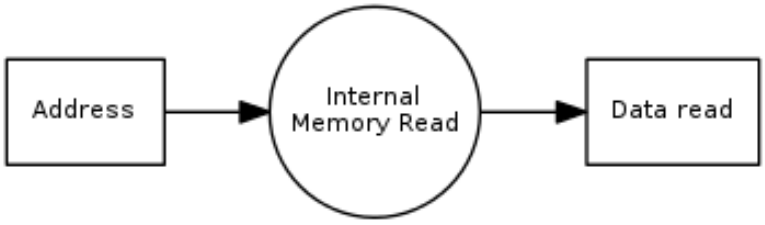

Figure 2.7: DFD Internal Memory Read Byte

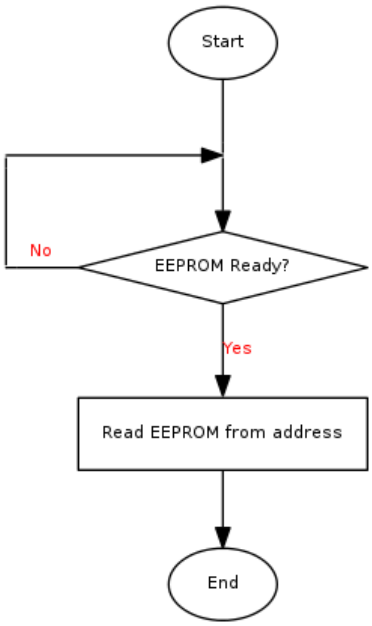

Figure 2.8: Flowchart Internal Memory Read Byte

2.4.1 Pengujian

Tabel 2.12 menampilkan Test Vector yang digunakan untuk menguji fungsi HAL Memory Internal Read Byte.

2.4.2 Implementasi

Tabel 2.9 menampilkan purwarupa dari implementasi fungsi HAL Memory Internal Read Byte.

Listing 2.4 menampilkan potongan program yang mengimplementasi fungsi HAL Memory Internal Read Byte.

```
1 uint8_t HAL_InternalReadByte(const uint8_t * addr)
```

<table>
<tr><th>Input
address</th><th>Output
Return Value</th></tr>
<tr><td colspan="2">Filled internal memory according to their address

0x000 - 0x00f: 0x00, 0x01, ..., 0x0f
0x010 - 0x01f: 0x10, 0x11, ..., 0x1f
...
0x0f0 - 0x0ff: 0xf0, 0xf1, ..., 0xff
...
0x100 - 0x1ff: 0x00, 0x01, ..., 0xff</td></tr>
<tr><td>0x000
0x001</td><td>= 0x00
= 0x01</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>0x0ff
0x100
0x101</td><td>= 0xff
= 0x00
= 0x01</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>0x1ff</td><td>= 0xff</td></tr>
</table>

Table 2.8: Test Vector Fungsi HAL Memory Internal Read Byte

Name	HAL_InternalReadByte
Input	alamat memory yang akan dibaca
Output	data hasil pembacaan (1 byte)

Table 2.9: Prototype Fungsi HAL Internal Memory Read Byte

```
2 {
3   while(!eeprom_is_ready());
4
5   return eeprom_read_byte(addr);
6 }
```

Listing 2.4: Listing Program Fungsi HAL Memory Internal Read Byte

2.5 Internal Memory Write Byte

Berfungsi menulis 1 byte data ke Memory EEPROM Internal. Gambar 2.9 menampilkan DFD dari fungsi HAL Internal Memory Write Byte.

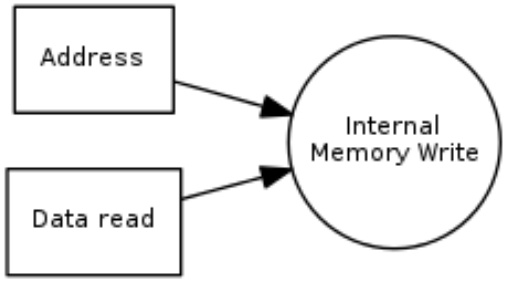

Figure 2.9: DFD Internal Memory Write Byte

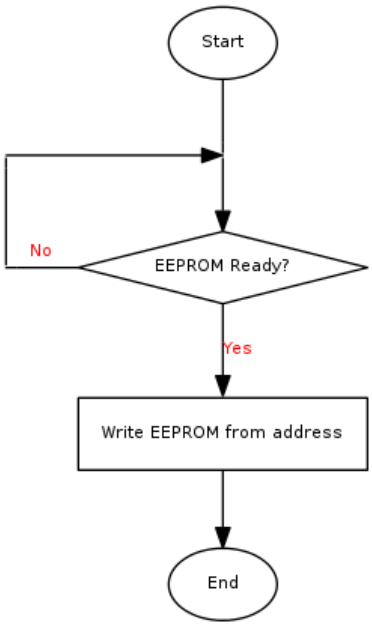

Figure 2.10: Flowchart Internal Memory Write Byte

2.5.1 Pengujian

Tabel 2.10 menampilkan Test Vector yang digunakan untuk menguji fungsi HAL Memory Internal Write Byte.

Input		Output
address	**data**	**Memory value @address**
0x000	0x00	= 0x00
...		
0x00f	0x0f	= 0x0f
0x010	0x10	= 0x10
...		
0x01f	0x1f	= 0x1f
...		
0x0ff	0xff	= 0xff
0x100	0x00	= 0x00
...		
0x1ff	0xff	= 0xff

Table 2.10: Test Vector Fungsi HAL Memory Internal Write Byte

2.5.2 Implementasi

Tabel 2.11 menampilkan purwarupa dari implementasi fungsi HAL Memory Internal Write Byte.

Name	HAL_InternalWriteByte
Input	• alamat memory yang akan ditulis • data yang akan ditulis (1 byte)
Output	-

Table 2.11: Prototype Fungsi HAL Internal Memory Write Byte

Listing 2.5 menampilkan potongan program yang mengimplementasi fungsi HAL Memory Internal Write Byte.

```
1 void HAL_InternalWriteByte(uint8_t * addr, uint8_t byte)
2 {
3   while(!eeprom_is_ready());
4
5   return eeprom_write_byte(addr, byte);
6 }
```

Listing 2.5: Listing Program Fungsi HAL Memory Internal Write Byte

2.6 External Memory Read Byte

Berfungsi membaca 1 byte data dari Memory EEPROM Eksternal.

Gambar 2.11 menampilkan DFD dari fungsi HAL External Memory Read Byte.

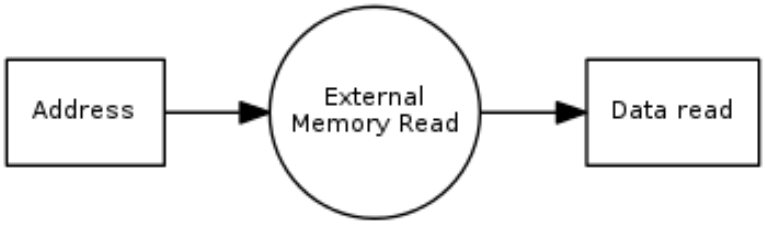

Figure 2.11: DFD External Memory Read Byte

2.6.1 Pengujian

Tabel 2.12 menampilkan Test Vector yang digunakan untuk menguji fungsi HAL Memory External Read Byte.

2.6.2 Implementasi

Tabel 2.13 menampilkan purwarupa dari implementasi fungsi HAL Memory External Read Byte.

Listing 2.6 menampilkan potongan program yang mengimplementasi fungsi HAL Memory External Read Byte.

```
; address r25:r24
; result r25(=0):r24
HAL_ExternalReadByte:
   push  r31
   push  r30
   push  r19
   push  r18
   push  r16
   push  r1
   push  r0
   mov   r31,r25
   mov   r30,r24
; Start
   rcall xereadlocal
; Done
```

<table>
<tr><th>Input
address</th><th>Output
Return Value</th></tr>
<tr><td colspan="2">Filled external memory according to their address

<table>
<tr><td>0x00000 - 0x000ff</td><td>[0x00, 0x01, ..., 0xff]</td></tr>
<tr><td></td><td>...</td></tr>
<tr><td>0x00100 - 0x00fff</td><td>[0x00, 0x01, ..., 0xff]x 0xE</td></tr>
<tr><td></td><td>...</td></tr>
<tr><td>0x01000 - 0x0ffff</td><td>[0x00, 0x01, ..., 0xff]x 0xEF</td></tr>
<tr><td></td><td>...</td></tr>
<tr><td>0x10000 - 0xfffff</td><td>[0x00, 0x01, ..., 0xff]x 0xEFF</td></tr>
</table></td></tr>
<tr><td>0x00000
0x00001</td><td>= 0x00
= 0x01</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>0x000ff
0x00100</td><td>= 0xff
= 0x00</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>0x00fff
0x01000</td><td>= 0xff
= 0x00</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>0x0ffff
0x10000</td><td>= 0xff
= 0x00</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>0xfffff</td><td>= 0xff</td></tr>
</table>

Table 2.12: Test Vector Fungsi HAL Memory External Read Byte

Name	HAL_ExternalReadByte
Input	alamat memory yang akan dibaca
Output	data hasil pembacaan (1 byte)

Table 2.13: Prototype Fungsi HAL External Memory Read Byte

```
16    clr   r25
17    mov   r24,r0
18    pop   r0
19    pop   r1
```

```
20    pop   r16
21    pop   r18
22    pop   r19
23    pop   r30
24    pop   r31
25    ret
26
27 ; address r31:r30
28 ; result r0 = XE(Z+)
29 xereadlocal:
30    rcall XEAddr
31    rcall XEStrt
32    clc
33    ldi     r19,0xA1
34    rcall XEEOut
35    rcall XE0Bit
36    rcall XEEIn
37    rcall XE1Bit
38    rcall XEStop
39    ret
```

Listing 2.6: Listing Program Fungsi HAL Memory External Read Byte

2.7 External Memory Write Byte

Berfungsi menulis 1 byte data ke Memory EEPROM External. Gambar 2.12 menampilkan DFD dari fungsi HAL External Memory Write Byte.

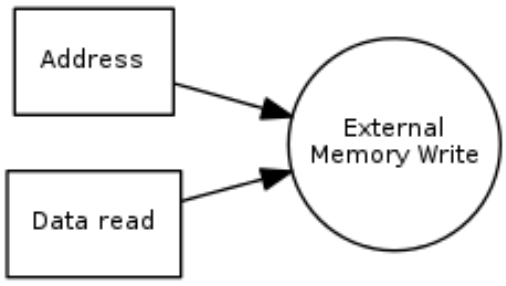

Figure 2.12: DFD External Memory Write Byte

Input		Output
address	**data**	**Memory value @address**
0x00000	0x00	= 0x00
...		
0x0000f	0x0f	= 0x0f
0x00010	0x10	= 0x10
...		
0x000ff	0x1f	= 0x1f
0x00100	0x10	= 0x10
...		
0x00fff	0xff	= 0xff
0x01000	0x00	= 0x00
...		
0x0ffff	0xff	= 0xff
0x10000	0x00	= 0x00
...		
0xfffff	0xff	= 0xff

Table 2.14: Test Vector Fungsi HAL Memory External Write Byte

2.7.1 Pengujian

Tabel 2.14 menampilkan Test Vector yang digunakan untuk menguji fungsi HAL Memory External Write Byte.

2.7.2 Implementasi

Tabel 2.15 menampilkan purwarupa dari implementasi fungsi HAL Memory External Write Byte.

Name	HAL_ExternalWriteByte
Input	• alamat memory yang akan ditulis • data yang akan ditulis (1 byte)
Output	-

Table 2.15: Prototype Fungsi HAL External Memory Write Byte

Listing 2.7 menampilkan potongan program yang mengimplementasi fungsi-HAL Memory External Write Byte.

```
; address r25:r24
; byte r23(=0):r22
xewrt:
   push r31
   push r30
   push r19
   push r18
   push r16
   push r1
   push r0
   mov  r31,r25
   mov  r30,r24
; Start
; address r31:r30
; result XE(Z+) = r22
   rcall xereadlocal
   cp r0,r22
   breq dontwrite
   rcall XEAddr
   mov  r19,r22
   rcall XEEOut
   rcall XE0Bit
   rcall XEStop
   rcall XEDly
dontwrite:
; Done
   pop  r0
   pop  r1
   pop  r16
   pop  r18
   pop  r19
   pop  r30
   pop  r31
   ret
```

Listing 2.7: Listing Program Fungsi HAL Memory External Write Byte

Chapter 3

Transmission Handler

Transmission Handler berfungsi menangani pengiriman dan penerimaan pesan APDU Command dan Response dari dan ke terminal. Pengiriman dan Penerimaan byte data pesan secara aktual dilakukan menggunakan fungsi HAL IO yang sesuai. Melalui antarmuka yang disediakan Transmission Handler, modul-modul pintarOS lainnya (terutama Command Handlers) tidak perlu mengetahui protokol yang digunakan pada proses pengiriman/penerimaan pesan APDU.

Tabel 3.1 menampilkan fungsi-fungsi yang terdapat pada modul Transmission beserta kegunaannya.

Nama Fungsi	**Kegunaan**
Get Byte	Menerima 1 byte data menggunakan protokol yang dipilih
Send Byte	Mengirim 1 byte data menggunakan protokol yang dipilih
Get Header	Menerima bagian header dari Command APDU
Get Data	Menerima bagian data dari Command APDU
Send Data	Mengirimkan bagian data dari Response APDU
Send SW	Mengirimkan bagian SW1 dan SW2 dari Response APDU

Table 3.1: Daftar antarmuka fungsi yang disediakan Transmission Handler

3.1 Get Byte

Berfungsi menerima 1 byte data dari IO menggunakan protokol komunikasi yang dipilih. Gambar 3.1 menampilkan DFD dari fungsi Transmission Get Byte (RxByte). Diagram alir fungsi kemudian ditampilkan pada Gambar **??**.

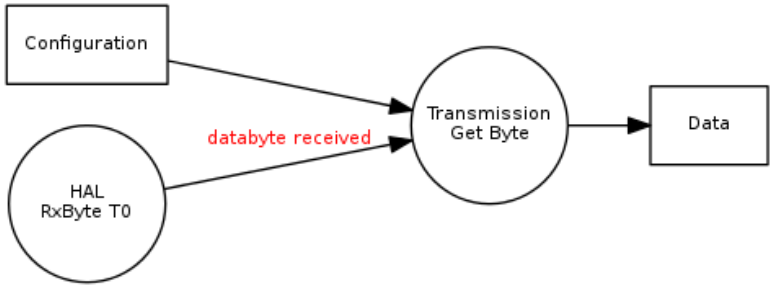

Figure 3.1: DFD Transmission GetByte

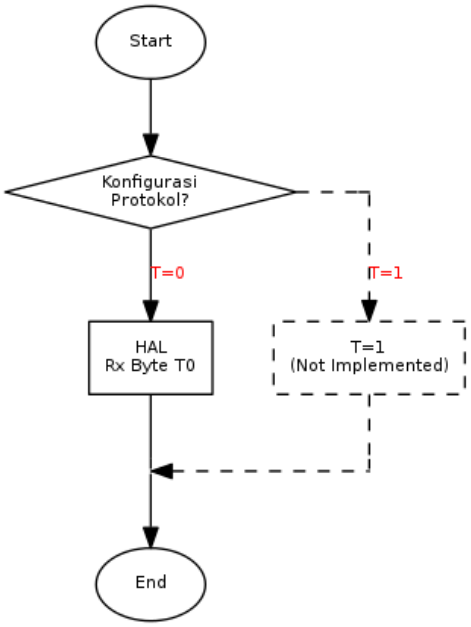

Figure 3.2: Flowchart Transmission Get Byte

3.1.1 Pengujian

Tabel 3.2 menampilkan Test Vector yang digunakan untuk menguji fungsi Transmission Get Byte.

Input	Output
Databyte (from terminal)	**Return Value**
0x00	= 0x00
0x01	= 0x01
...	
...	
0xff	= 0xff

Table 3.2: Test Vector Fungsi Transmission GetByte

3.1.2 Implementasi

Tabel 3.3 menampilkan purwarupa dari implementasi fungsi Transmission Get Byte.

Name	Transmission_GetByte
Input	
Output	Data yang diterima (1 byte)

Table 3.3: Prototype Fungsi Transmission Get Byte

Listing 3.1 menampilkan potongan program yang mengimplementasi fungsi Transmission Get Byte.

```
1 uint8_t Transmission_GetByte()
2 {
3   /* if (config.proto == 0) */
4     return HAL_RxByteT0();
5 }
```

Listing 3.1: Listing Program Fungsi Transmisison Get Byte

3.2 Send Byte

Berfungsi mengirimkan 1 byte data menggunakan protokol komunikasi yang dipilih. Gambar 3.3 menampilkan DFD dari fungsi Transmission Send Byte (RxByte). Diagram alir fungsi kemudian ditampilkan pada Gambar 3.4.

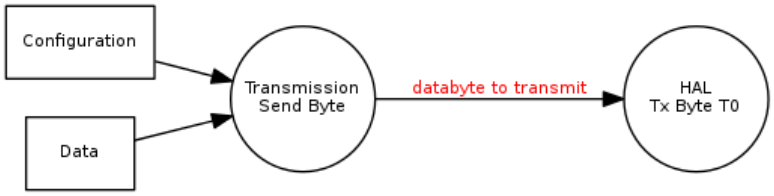

Figure 3.3: DFD Transmission Send Byte

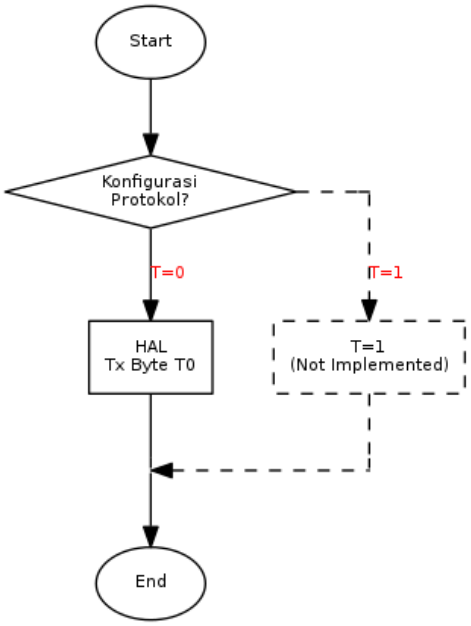

Figure 3.4: Flowchart Transmission Send byte

3.2.1 Pengujian

Tabel 3.4 menampilkan Test Vector yang digunakan untuk menguji fungsi Transmission Send Byte.

3.2.2 Implementasi

Tabel 3.5 menampilkan purwarupa dari implementasi fungsi Transmission Send Byte.

Listing 3.2 menampilkan potongan program yang mengimplementasi fungsi Transmission Send Byte.

```
1 void Transmission_SendByte(uint8_t databyte)
2 {
```

Input	Output
Data	**Databyte (to terminal)**
0x00	= 0x00
0x01	= 0x01
...	
...	
0xff	= 0xff

Table 3.4: Test Vector Fungsi Transmission SendByte

Name	Transmission_SendByte
Input	Data yang akan dikirimkan (1 byte)
Output	-

Table 3.5: Prototype Fungsi Transmission Send Byte

```
3   /* if (config.protocol == 0) */
4     HAL_TxByteT0(databyte);
5 }
```

Listing 3.2: Listing Program Fungsi Transmisison Send Byte

3.3 Get Header

Berfungsi menerima 5 byte pertama dari command APDU (header). Header ini kemudian disimpan pada variable array *header* yang terdapat pada *Command Handlers*. Gambar 3.5 menampilkan DFD dari fungsi Get Header. Diagram alir fungsi kemudian ditampilkan pada Gambar 3.6.

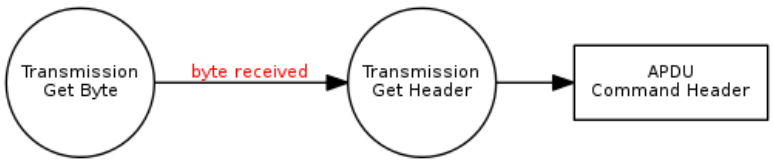

Figure 3.5: DFD Transmission Get Header

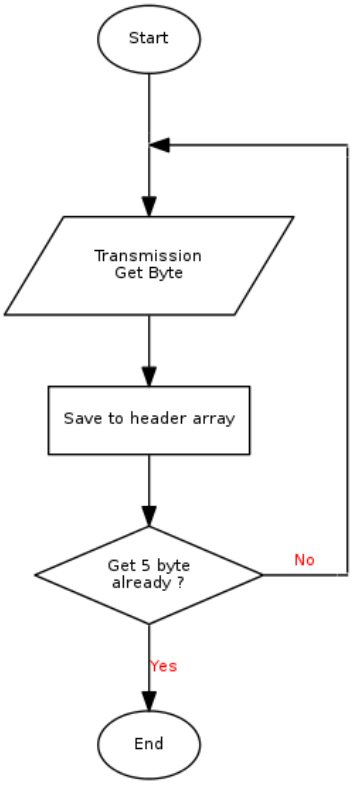

Figure 3.6: Flowchart Transmission Get Header

3.3.1 Pengujian

Input	Output
Databyte	**header**
[0x00,0x01,0x02,0x03,0x04,..,0x0f]	= [0x00,0x01,0x02,0x03,0x04]

Table 3.6: Test Vector Fungsi Transmission GetHeader

Tabel 3.6 menampilkan Test Vector yang digunakan untuk menguji fungsi Transmission Get Header.

3.3.2 Implementasi

Tabel 3.7 menampilkan purwarupa dari implementasi fungsi Transmission Get Header.

Listing 3.3 menampilkan potongan program yang mengimplementasi fungsi Get Header.

Name	Transmission_GetHeader
Input	-
Output	-

Table 3.7: Prototype Fungsi Transmission Get Header

```
1 void Transmission_GetHeader()
2 {
3   uint8_t i;
4
5   for (i = 0; i < 5; ++i)
6     {
7       header[i] = Transmission_GetByte();
8     }
9 }
```

Listing 3.3: Listing Program Fungsi Get Header

3.4 Get Data

Berfungsi menerima bagian data dari Command APDU, yang diperoleh dari dari IO, untuk digunakan oleh *command handler*. Gambar 3.7 menampilkan DFD dari fungsi Get Data. Diagram alir fungsi kemudian ditampilkan pada Gambar 3.8.

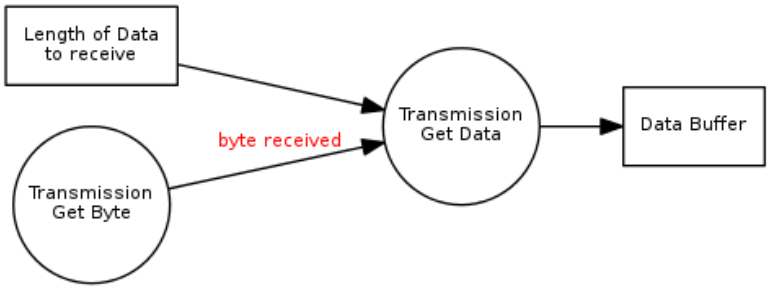

Figure 3.7: DFD Transmission Get Data

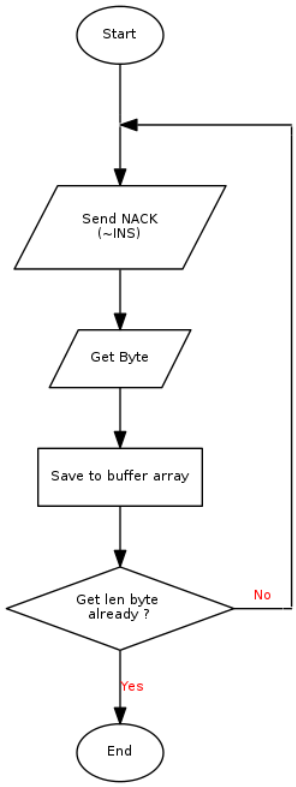

Figure 3.8: Flowchart Get Data

3.4.1 Pengujian

Tabel 3.8 menampilkan Test Vector yang digunakan untuk menguji fungsi Transmission Get Data.

3.4.2 Implementasi

Tabel 3.9 menampilkan purwarupa dari implementasi fungsi Transmission Get Data.

Listing 3.4 menampilkan potongan program yang mengimplementasi fungsi Get Data.

```
1 void Transmission_GetData(uint8_t * dst, uint8_t len)
2 {
3   while ( len>0 )
4     {
5       Transmission_SendNACK();
6       *(dst++) = Transmission_GetByte();
```

Input		Output
Databyte (from terminal)	**length**	**output buffer**
[0x00,0x01,0x02,0x03,0x04,..,0x0f]	1	[=0x00]
[0x00,0x01,0x02,0x03,0x04,..,0x0f]	2	[=0x00,0x01]
[0x00,0x01,0x02,0x03,0x04,..,0x0f]	3	[=0x00,0x01,0x02]
[0x00,0x01,0x02,0x03,0x04,..,0x0f]	4	[=0x00,0x01,0x02,0x03]
[0x00,0x01,0x02,0x03,0x04,..,0x0f]	5	[=0x00,0x01,0x02,0x03,0x04]

Table 3.8: Test Vector Fungsi Transmission GetData

Name	Transmission_GetData
Input	• Alamat buffer dimana data akan disimpan • Panjang data yang akan diterima
Output	-

Table 3.9: Prototype Fungsi Get Data

```
7       len--;
8     }
9   }
```

Listing 3.4: Listing program fungsi Get Data

3.5 Send Data

Berfungsi mengirimkan bagian data dari Response APDU. Digunakan oleh *command handler*. Gambar 3.9 menampilkan DFD dari fungsi Get Header. Diagram alir fungsi kemudian ditampilkan pada Gambar 3.10.

3.5.1 Pengujian

Tabel 3.10 menampilkan Test Vector yang digunakan untuk menguji fungsi Transmission Send Data.

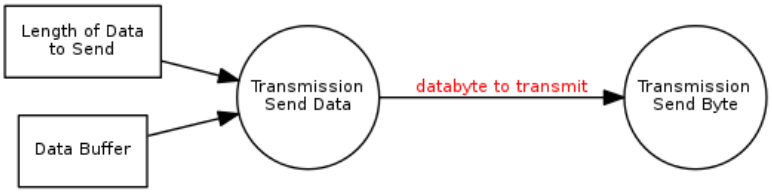

Figure 3.9: DFD Transmission Send Data

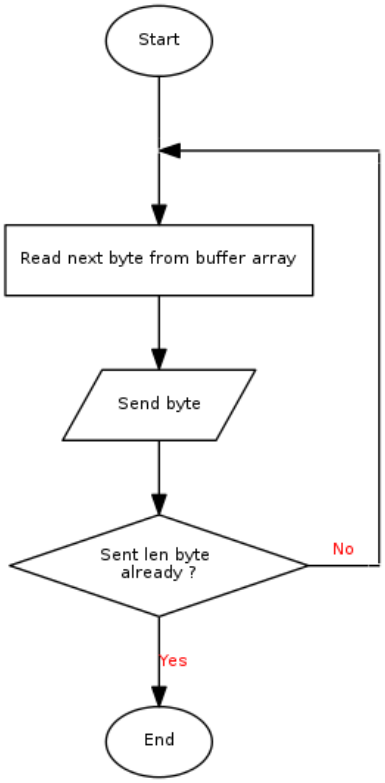

Figure 3.10: Flowchart Transmission Send Data

3.5.2 Implementasi

Tabel 3.11 menampilkan purwarupa dari implementasi fungsi Transmission Send Data.

Listing 3.5 menampilkan potongan program yang mengimplementasi fungsi Send Data.

```
1 void Transmission_SendData(uint8_t * src, uint8_t len)
2 {
3   while ( len>0 )
```

Input		Output
input buffer	**length**	**databyte (to terminal)**
[0x00,0x01,0x02,0x03,0x04,..,0x0f]	1	= [0x00]
[0x00,0x01,0x02,0x03,0x04,..,0x0f]	2	= [0x00,0x01]
[0x00,0x01,0x02,0x03,0x04,..,0x0f]	3	= [0x00,0x01,0x02]
[0x00,0x01,0x02,0x03,0x04,..,0x0f]	4	= [0x00,0x01,0x02,0x03,]
[0x00,0x01,0x02,0x03,0x04,..,0x0f]	5	= [0x00,0x01,0x02,0x03,0x04]

Table 3.10: Test Vector Fungsi Transmission SendData

Name	Transmission_SendData
Input	• Alamat dari buffer data yang akan dikirim • Panjang data yang akan dikirimkan
Output	-

Table 3.11: Prototype Fungsi Send Data

```
4     {
5       Transmission_SendByte( *(src++) );
6       len--;
7     }
8   }
```

Listing 3.5: Listing program fungsi Send Data

3.6 Send SW

Berfungsi mengirimkan Status Word dari Response APDU. Status Word yang akan dikirimkan diperoleh dari variable *sw* yang telah di-set oleh *Response_SetSW*. Gambar 3.11 menampilkan DFD dari fungsi Send SW. Diagram alir fungsi kemudian ditampilkan pada Gambar 3.12.

3.6.1 Pengujian

Tabel 3.12 menampilkan Test Vector yang digunakan untuk menguji fungsi Transmission Send SW.

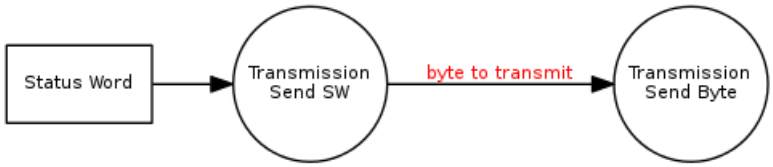

Figure 3.11: DFD Transmission Send SW

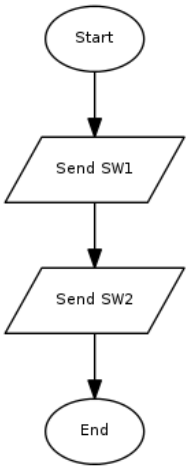

Figure 3.12: Flowchart Send SW

3.6.2 Implementasi

Tabel 3.13 menampilkan purwarupa dari implementasi fungsi Transmission Send SW.

Listing 3.6.2 menampilkan potongan program yang mengimplementasi fungsi Transmission Send SW.

```
1 void Transmission_SendSW()
2 {
3   Transmission_SendByte( ( (uint8_t)(sw>>8) & 0xFF) );
4   Transmission_SendByte( (uint8_t)(sw & 0xFF) );
5 }
```

Listing program fungsi Send SW

Input	**Output**
SW	**output buffer**
0x9000	= [0x90,0x00]
0x63C1	= [0x63,0xC1]

Table 3.12: Test Vector Fungsi Transmission Send SW

Name	Transmission_SendSW
Input	-
Output	-

Table 3.13: Prototype Fungsi Transmission Send SW

Chapter 4

Memory Handlers

Memory Handlers berfungsi menangani pembacaan dan penulisan data ke media penyimpanan secara umum. Pembacaan dan penulisan secara aktual dilakukan dengan memanggil fungsi HAL Memory yang sesuai. Memory menyediakan sebuah alamat virtual untuk menyimpan dan memperoleh data tanpa harus mengetahui lokasi data sebenarnya pada sistem. Konfigurasi virtual address yang digunakan tergantung pada platform smartcard yang digunakan. Gambar **??** menampilkan contoh konfigurasi virtual memory pada platform funcard.

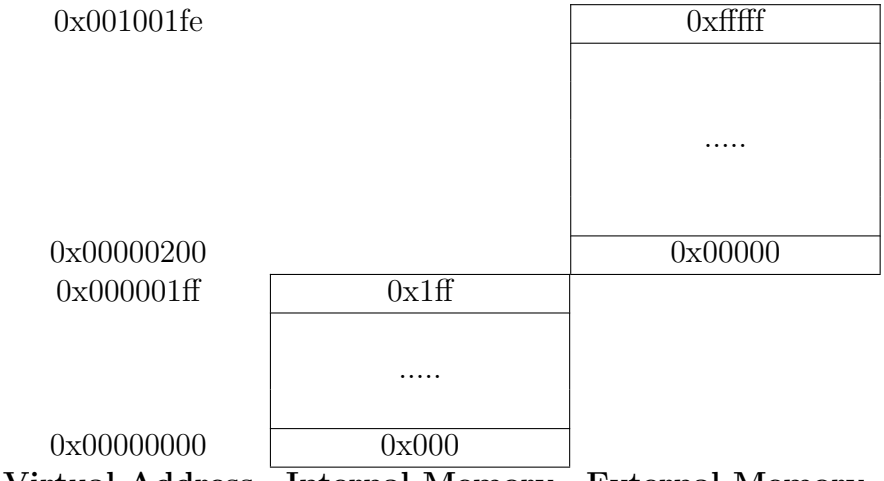

Table 4.1: Contoh konfigurasi virtual address pada platform funcard

Pada file konfigurasi, ukuran (batas atas) virtual memory dinyatakan sebagai MEMORY_SIZE, dan batas antara bagian virtual memory yang menggunakan Internal dan Eksternal Memory dinyatakan sebagai INTERNAL_MEMORY_SIZE.

Memory Handlers menyediakan sejumlah interface yang dapat digunakan bagian pintarOS lainnya (terutama File System) sebagai metode pembacaan/penulisan sebagaimana didaftarkan pada Tabel **??**.

Nama Fungsi	**Kegunaan**
Read Byte	Membaca 1 byte data dari alamat virtual
Write Byte	Menulis 1 byte data ke alamat virtual
Read Block	Membaca 1 block data dari alamat virtual
Write Block	Menulis 1 block data ke alamat virtual

Table 4.2: Daftar antarmuka fungsi yang disediakan Memory Handlers

4.1 Memory Read Byte

Berfungsi membaca 1 byte data dari Memory EEPROM (internal eksternal). Gambar 4.1 menampilkan DFD dari fungsi Memory Read Byte (ReadByte). Diagram alir fungsi kemudian ditampilkan pada Gambar 4.2. Fungsi ini pertama akan mencari lokasi memory yang akan dibaca dari alamat yang diberikan, apakah terdapat pada memory internal atau eksternal. Setelah mengetahui lokasi memory yang akan dibaca, fungsi ini akan memanggil fungsi HAL yang sesuai yang kemudian akan membaca byte data secara aktual dari lokasi memory.

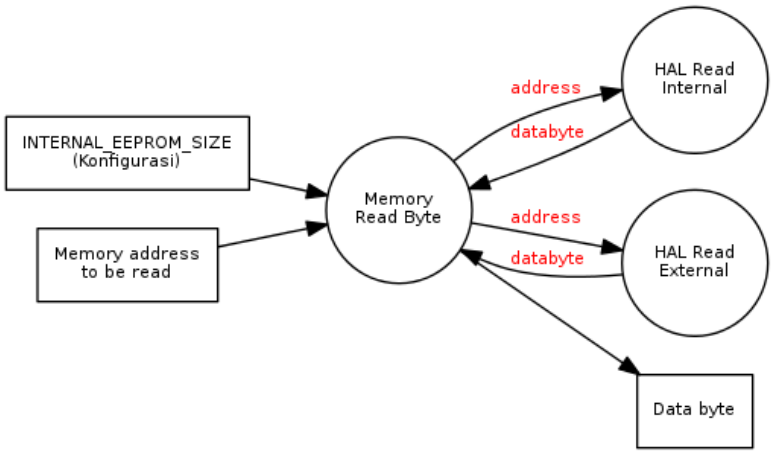

Figure 4.1: DFD Memory Read Byte

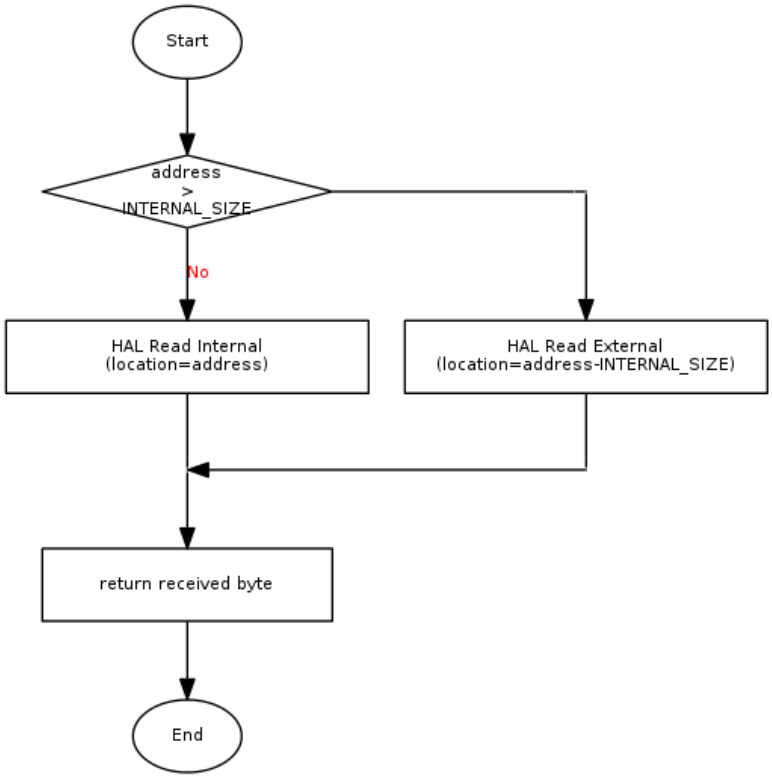

Figure 4.2: Flowchart Memory Read Byte

4.1.1 Pengujian

Tabel 4.3 menampilkan Test Vector yang digunakan untuk menguji fungsi Memory Read Byte.

4.1.2 Implementasi

Tabel 4.4 menampilkan purwarupa dari implementasi fungsi Memory Read Byte

Listing 4.1 menampilkan potongan program yang mengimplementasi fungsi Memory Read Byte.

```
1 uint8_t Memory_ReadByte(uint16_t address)
2 {
3   if( address < INTERNAL_EEPROM_SIZE )
4     {
5       return readinternalmem(address);
6     }
```

<table>
<tr><th>Input
address</th><th>Output
Return Value</th></tr>
<tr><td colspan="2">Filled internal and external memory according to their virtual address

0x0000 - 0x000f: 0x00, 0x01, ..., 0x0f
0x0010 - 0x001f: 0x10, 0x11, ..., 0x1f
...
0x00f0 - 0x00ff: 0xf0, 0xf1, ..., 0xff
0x0100 - 0x010f: 0x00, 0x01, ..., 0x0f
0x0110 - 0x011f: 0x10, 0x11, ..., 0x1f
...
0xfff0 - 0xffff: 0xf0, 0xf1, ..., 0xff</td></tr>
<tr><td>0x0000
0x0001</td><td>= 0x00
= 0x01</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>0x00ff
0x0100
0x0101</td><td>= 0xff
= 0x00
= 0x01</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>0x01ff</td><td>= 0xff</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>0xff00
0xff01</td><td>= 0x00
= 0x01</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>0xffff</td><td>= 0xff</td></tr>
<tr><td colspan="2">...</td></tr>
<tr><td>0xff</td><td>= 0xff</td></tr>
</table>

Table 4.3: Test Vector Fungsi Memory Read Byte

```
7   else
8     {
9       return readexternalmem(address - INTERNAL_EEPROM_SIZE);
10    }
11  }
```

Listing 4.1: Listing Program Fungsi Memory Read Byte

Name	Memory_ReadByte
Input	alamat memory virtual yang akan dibaca
Output	data hasil pembacaan (1 byte)

Table 4.4: Prototype Fungsi Memory Read Byte

4.2 Memory Write Byte

Berfungsi menulis 1 byte data ke Memory EEPROM (internal eksternal). Gambar 4.3 menampilkan DFD dari fungsi Memory Write Byte (WriteByte). Diagram alir fungsi kemudian ditampilkan pada Gambar 4.4. Fungsi ini pertama akan mencari lokasi memory yang akan ditulis dari alamat yang diberikan, apakah terdapat pada memory internal atau eksternal. Setelah mengetahui lokasi memory yang akan ditulis, fungsi ini akan memanggil sebuah fungsi HAL yang sesuai yang akan secara aktual menulis byte data ke lokasi memory.

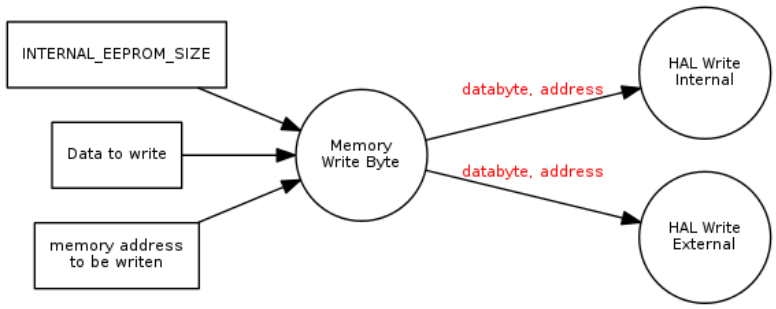

Figure 4.3: DFD Memory Write Byte

4.2.1 Pengujian

Tabel 4.5 menampilkan Test Vector yang digunakan untuk menguji fungsi Memory Write Byte.

4.2.2 Implementasi

Tabel 4.6 menampilkan purwarupa dari implementasi fungsi Memory Write Byte

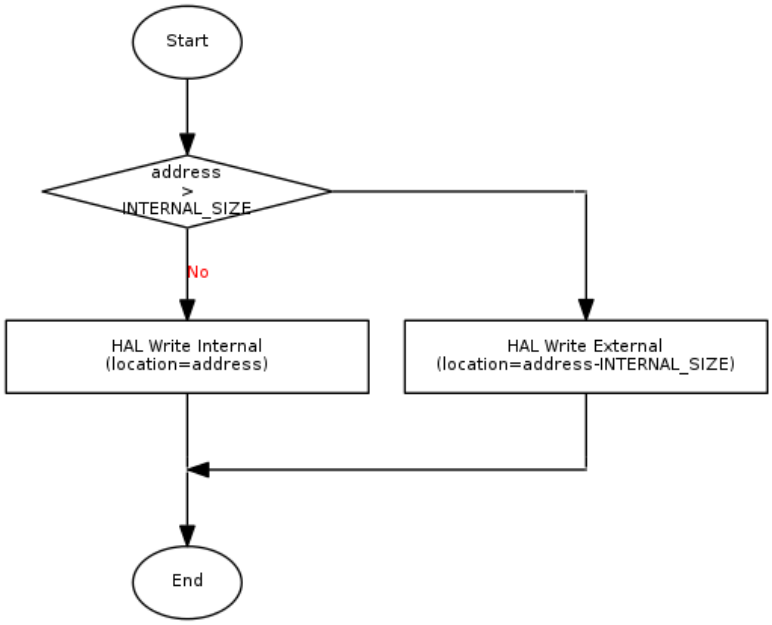

Figure 4.4: Flowchart Memory Write Byte

Listing 4.2 menampilkan potongan program yang mengimplementasi fungsi Memory Write Byte.

```
1  void Memory_WriteByte(uint16_t address, uint8_t databyte)
2  {
3    if( address < INTERNAL_EEPROM_SIZE )
4      {
5        writeinternalmem(address, databyte);
6      }
7    else
8      {
9        writeexternalmem(address - INTERNAL_EEPROM_SIZE, databyte);
10     }
11 }
```

Listing 4.2: Listing Program Fungsi Memory Write Byte

4.3 Memory Read Block

Berfungsi membaca 1 block data dari Memory EEPROM (internal eksternal). Gambar 4.5 menampilkan DFD dari fungsi Memory Read Block (ReadBlock). Diagram alir fungsi kemudian ditampilkan pada Gambar 4.6. Fungsi

Input		**Output**
address	**data**	**Memory value @address**
0x0000	0x00	= 0x00
...		
0x000f	0x0f	= 0x0f
0x0010	0x10	= 0x10
...		
0x001f	0x1f	= 0x1f
...		
0x00ff	0xff	= 0xff
0x0100	0x00	= 0x00
...		
0x01ff	0xff	= 0xff
...		
0x0fff	0xff	= 0xff
0x1f00	0x00	= 0x00
...		
0x1fff	0xff	= 0xff
...		
0xffff	0xff	= 0xff

Table 4.5: Test Vector Fungsi Memory Write Byte

Name	Memory_WriteByte
Input	• alamat memory yang akan ditulis • data yang akan ditulis (1 byte)
Output	-

Table 4.6: Prototype Fungsi Memory Write Byte

ini akan membaca memory dimulai dari alamat memory yang diberikan berturut-turut sepanjang *len* byte dengan memanggil fungsi Read Byte.

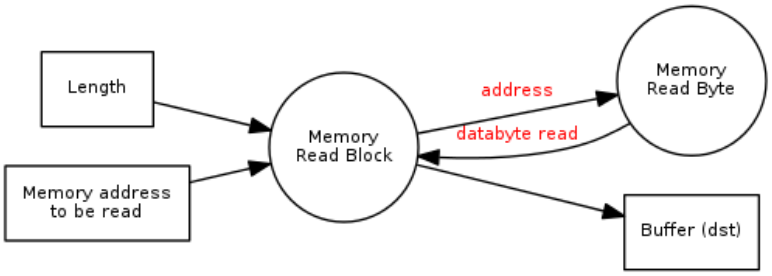

Figure 4.5: DFD Memory Read Block

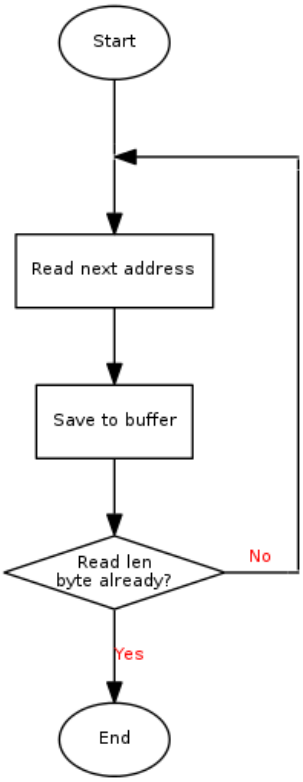

Figure 4.6: Flowchart Memory Read Block

4.3.1 Pengujian

Tabel 4.7 menampilkan Test Vector yang digunakan untuk menguji fungsi Memory Read Block.

<table>
<tr><th colspan="2">Input</th><th>Output</th></tr>
<tr><th>address</th><th>length</th><th>Output buffer</th></tr>
<tr><td colspan="3">Filled internal and external memory according to their virtual address

0x0000 - 0x000f | 0x00, 0x01, ..., 0x0f
0x0010 - 0x001f | 0x10, 0x11, ..., 0x1f
... | ...
0x00f0 - 0x00ff | 0xf0, 0xf1, ..., 0xff
0x0100 - 0x010f | 0x00, 0x01, ..., 0x0f
0x0110 - 0x011f | 0x10, 0x11, ..., 0x1f
... | ...
0xfff0 - 0xffff | 0xf0, 0xf1, ..., 0xff</td></tr>
<tr><td>0x0000</td><td>0x0001</td><td>= [0x00]</td></tr>
<tr><td>0x0001</td><td>0x000e</td><td>= [0x01, 0x02, ..., 0x0f]</td></tr>
<tr><td>0x0010</td><td>0x00ef</td><td>= [0x10, 0x11, ..., 0xff]</td></tr>
<tr><td>0x0100</td><td>0x0eff</td><td>= [0x00, 0x01, ..., 0xff]x15</td></tr>
<tr><td>0x1000</td><td>0xefff</td><td>= [0x00, 0x01, ..., 0xff]x239</td></tr>
</table>

Table 4.7: Test Vector Fungsi Memory Read Block

Name	Memory_ReadBlock
Input	• awal alamat virtual memory yang akan dibaca • panjang data yang akan dibaca • alamat buffer dimana data akan disimpan
Output	jumlah byte yang terbaca

Table 4.8: Prototype Fungsi Memory Read Block

4.3.2 Implementasi

Tabel 4.8 menampilkan purwarupa dari implementasi fungsi Memory ReadBlock. Listing 4.3 menampilkan potongan program yang mengimplementasi fungsi Memory Read Block.

```
int Memory_ReadBlock(uint16_t address, uint16_t size, uint8_t *
    databyte)
{
  uint16_t count;

  for( count=0; count < size; count++)
    {
      *(databyte+count) = Memory_ReadByte(address+count);
    }

  return count;
}
```

Listing 4.3: Listing Program Fungsi Memory Read Block

4.4 Memory Write Block

Berfungsi menulis 1 block data ke Memory EEPROM (internal eksternal). Gambar 4.7 menampilkan DFD dari fungsi Memory Write Block (Write-Block). Diagram alir fungsi kemudian ditampilkan pada Gambar 4.8. Fungsi ini akan menulis data dari buffer ke memory dimulai dari alamat memory yang diberikan berturut-turut sepanjang *len* byte dengan memanggil fungsi Memory Write Byte.

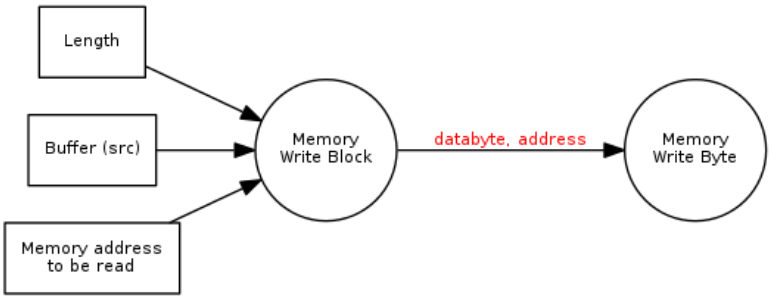

Figure 4.7: DFD Memory Write Block

4.4.1 Pengujian

Tabel 4.9 menampilkan Test Vector yang digunakan untuk menguji fungsi Memory Write Block.

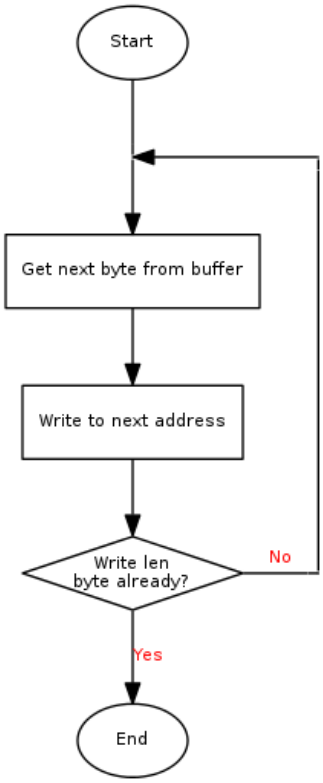

Figure 4.8: Flowchart Memory Write Block

Input			Output
address	**data**	**length**	**Memory value @address→address+length**
0x0000	[0x00]	0x0001	= [0x00]
0x0001	[0x01, ..., 0x0f]	0x000e	= [0x01, ..., 0x0f]
0x0010	[0x10, ..., 0xff]	0x00ef	= [0x10, ..., 0xff]
0x0100	[0x00, ..., 0xff]X15	0x0eff	= [0x00, ..., 0xff]X15
0x1000	[0x00, ..., 0xff]X239	0xefff	= [0x00, ..., 0xff]X239

Table 4.9: Test Vector Fungsi Memory Write Block

4.4.2 Implementasi

Tabel 4.10 menampilkan purwarupa dari implementasi fungsi Memory Write Block. Listing 4.4 menampilkan potongan program yang mengimplementasi fungsi Memory Write Block.

```
1 int Memory_WriteBlock(uint16_t address, uint16_t size, uint8_t *
    databyte)
```

Name	Memory Write Block
Input	• awal alamat memory virtual yang akan ditulis • alamat buffer data yang akan ditulis
Output	jumlah byte yang berhasil ditulis

Table 4.10: Prototype Fungsi Memory Write Block

```
2  {
3    uint16_t count;
4
5    for( count=0; count < size; count++)
6      {
7        Memory_WriteByte(address+count, *(databyte+count));
8      }
9
10   return count;
11 }
```

Listing 4.4: Listing Program Fungsi Memory Write Block

Chapter 5

State Manager

Dalam penggunaannya smart card bekerja dalam serangkaian Command, dan seringkali diantara command tersebut terdapat rangkaian command yang saling berkaitan. Command yang saling berkaitan tersebut umumnya bekerja menggunakan informasi yang sama, dimana command yang datang kemudian menggunakan informasi yang dihasilkan oleh command sebelumnya. State Manager berfungsi menyimpan informasi-informasi yang berhubungan tersebut dalam sebuah state.

Sebagai contoh, command Read Binary akan membaca data dari sebuah file yang telah dipilih sebelumnya menggunakan command Select. Informasi mengenai File yang sedang dipilih ini harus bersifat persistent selama sesi smartcard berlangsung, namun tidak setelahnya. Karenanya informasi ini hanya perlu disimpan pada RAM, dan tidak pada EEPROM.

Terdapat beberapa contoh rangkaian command lainnya ditampilkan pada 5.1, yang nantinya akan digunakan untuk menyusun daftar data/informasi yang perlu disimpan di dalam state.

No	Command	Penjelasan
1	$Select \rightarrow Select$	Test Select akan mencari dan memilih File yang terdapat 1 tingkat didalamnya (direct child), 1 tingkat diatasnya (parent), atau 1 tingkat didalam file parent (sibling) dari file yang telah dipilih sebelumnya menggunakan Select
2	$Select \rightarrow ReadBinary$	Read Binary akan membaca data binary dari file yang telah dipilih sebelumnya menggunakan Select
3	$Select \rightarrow UpdateBinary$	Update Binary akan menulis data binary ke file yang telah dipilih sebelumnya menggunakan Select
4	$Select \rightarrow CreateFile$	Create File akan membuat sebuah file baru didalam (sebagai child) file yang telah dipilih sebelumnya menggunakan Select -apabila file terpilih merupakan DF- ataupun file yang 1 tingkat diatasnya (parent) -apabila file terpilih merupakan EF-.
5	$Select \rightarrow DeleteFile$	Delete File akan menghapus sebuah File yang berada 1 tingkat dibawah (child) file yang telah dipilih sebelumnya menggunakan Select -apabila file terpilih merupakan DF- ataupun file lainnya yang berada 1 tingkat dibawah file parent (sibling) -apabila file terpilih merupakan EF-.
6	$Verify \rightarrow ReadBinary$	Read Binary hanya akan membaca data binary dari sebuah file terproteksi PIN apabila pengguna telah terverifikasi sebelumnya menggunakan command Verify.
7	$Verify \rightarrow UpdateBinary$	Update Binary hanya akan menulis data binary ke sebuah file terproteksi PIN apabila pengguna telah terverifikasi sebelumnya menggunakan command Verify.
8	$GetChallenge \rightarrow ExternalAuth$	External Auth akan melakukan proses otentikasi menggunakan challenge terakhir yang dibuat menggunakan command Get Challenge
9	$ExternalAuth \rightarrow ReadBinary$	Read Binary hanya akan membaca data binary dari sebuah file terproteksi Auth apabila pengguna telah terotentikasi sebelumnya menggunakan command External Authenticate.
10	$ExternalAuth \rightarrow UpdateBinary$	Update Binary hanya akan menulis data binary ke sebuah file terproteksi Auth apabila pengguna telah terotentikasi sebelumnya menggunakan command External Authenticate.

Table 5.1: Contoh-contoh rangkaian command yang saling berkaitan

Berdasarkan contoh-contoh rangkaian command yang saling terkait pada Tabel 5.1 dapat diperoleh daftar data/informasi yang perlu untuk disimpan dalam state adalah sebagai berikut :

1. File Terpilih

   File Terpilih akan menyimpan alamat block dari header file yang sedang dipilih pada Memory, sebagaimana diterangkan pada *File System*. Karena alamat block membutuhkan ruang penyimpanan sebesar 2 byte (16 bit), demikian pula pada File Terpilih akan menggunakan tipe data unsigned integer 16 bit.

2. Security State Aktif

   Security State akan menyimpan status keamanan yang telah dicapai oleh pengguna melalui proses verifikasi identitas. Terdapat 2 metode verifikasi identitas yang digunakan pada pintarOS, yaitu menggunakan PIN dan eksternal authentication yang menggunakan Cryptography Key.

   Pada metode verifikasi menggunakan PIN, pengguna harus memasukkan PIN yang sama dengan yang tersimpan pada Memory SmartCard pada alamat Memory yang didefinisikan pada file konfigurasi sebagai PIN_ADDR. Terdapat batasan percobaan yang dapat dilakukan oleh pengguna yang didefinisikan pada file konfigurasi sebagai PIN_MAX_RETRIES yang secara default bernilai 3.

   Pada metode verifikasi External Authentication, pengguna harus mengembalikan 1 block data acak (challenge) yang telah dienkripsi menggunakan kunci yang dimiliki pengguna. Smart Card kemudian akan membandingkannya dengan hasil enkripsi terhadap data acak (challenge) yang sama menggunakan kunci yang tersimpan pada Memory SmartCard. Apabila kunci yang digunakan pengguna adalah sama dengan kunci yang digunakan Smart Card, maka data hasil enkripsi harusnya akan menjadi sama. Sebaliknya apabila kunci yang digunakan berbeda, akan menghasilkan hasil enkripsi yang berbeda. Dengan 7 kunci berbeda yang tersimpan pada alamat Memory KEY1_ADDR hingga KEY7_ADDR, berarti Smart Card dapat melakukan otentikasi hingga 7 pengguna (peran pengguna).

   Setiap bit pada securityState menyimpan informasi untuk satu metode verifikasi. Bit pertama (0) akan menyimpan informasi mengenai verifikasi PIN. Bit sisanya (7-1) dapat digunakan untuk menyimpan informasi verifikasi menggunakan authentikasi eksternal untuk setiap (7) Key. Pembagian fungsi bit ini ditunjukkan pada Gambar 5.1.

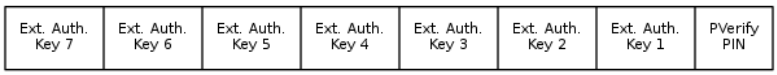

Figure 5.1: Pembagian Fungsi Bit Security State

Setiap verifikasi yang berhasil akan mengubah nilai bit yang bersesuaian menjadi 1 (set). Sebaliknya ketika verifikasi gagal, nilai bit yang bersesuaian akan menjadi 0 (reset).

3. Challenge paling akhir

   Challenge akan menyimpan data acak yang akan digunakan saat melakukan external authentication. Merupakan sebuah array integer 8 bit berukuran 1 block untuk algoritma kriptografi yang digunakan.

Ketiga data/informasi ini akan disimpan sebagai sebuah state didalam sebuah struktur data dengan tipe state_struct yang definisinya diberikan pada Listing 5.1.

```
1 struct state_struct
2 {
3   uint16_t       current;    ///< pointer to current DF header
4   uint8_t        securityState; ///< security state currently active
5   uint8_t        challenge[CRYPT\_BLOCK\_LEN];
6 };
```

Listing 5.1: Definisi tipe struktur data state_struct

5.1 State Initialization

Berfungsi menginisialisasi state manager. dipanggil setiap kali memulai sesi baru (ketika smartcard dimasukkan ke reader). Gambar 5.2 menampilkan DFD dari fungsi State Initialization. Diagram alir fungsi kemudian ditampilkan pada Gambar 5.3.

5.1.1 Pengujian

Tabel 5.2 menampilkan Test Vector yang digunakan untuk menguji fungsi State Init.

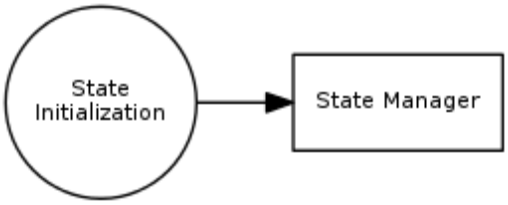

Figure 5.2: DFD State Initialization

Output
state
currentFile = 0 currentRecord = 0 securityState = 0x00

Table 5.2: Test Vector Fungsi State Init

5.1.2 Implementasi

Tabel 5.3 menampilkan purwarupa dari implementasi fungsi State Initialization.

Name	State_Init
Input	-
Output	Result Status

Table 5.3: Prototype Fungsi State Initialization

Listing 5.2 menampilkan potongan program yang mengimplementasi fungsi State Initialization

```
1 int State_Init()
2 {
3   state_mng.current = 0;
4   state_mng.securityState = 0;
5
6   return STATE_OK;
7 }
```

Listing 5.2: Listing Program Fungsi State Initialization

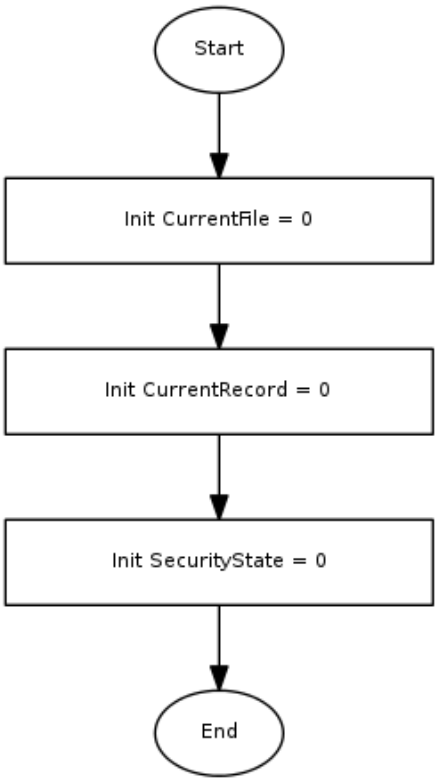

Figure 5.3: Flowchart State Initialization

5.2 Set Current File

Berfungsi Sebagai setter untuk statemember current file. Gambar 5.4 menampilkan DFD dari fungsi Set Current File. Diagram alir fungsi kemudian ditampilkan pada Gambar 5.5.

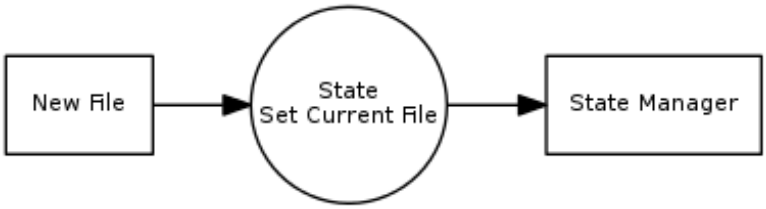

Figure 5.4: DFD Set Current File

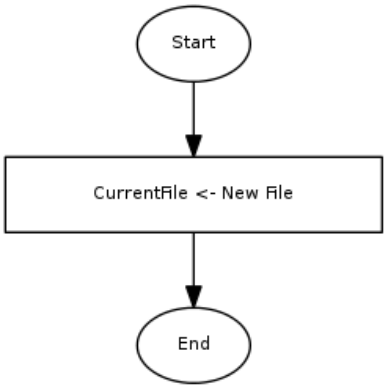

Figure 5.5: Flowchart Set Current File

Input	Output
New File	**State**
0x0000	currentFile = 0x0000
0x000f	currentFile = 0x000f
0x00ff	currentFile = 0x00ff
0x0fff	currentFile = 0x0fff
0xffff	currentFile = 0xffff

Table 5.4: Test Vector Fungsi State Set Current File

5.2.1 Pengujian

Tabel 5.4 menampilkan Test Vector yang digunakan untuk menguji fungsi State Set Current File.

5.2.2 Implementasi

Name	State_SetCurrent
Input	new current file
Output	Result Status

Table 5.5: Prototype Fungsi Set Current File

Listing 5.3 menampilkan potongan program yang mengimplementasi fungsi Set Current File.

```
int State_SetCurrent(uint16_t newFile)
{
  state_mng.current = newFile;
  state_mng.currentRecord = 0;
  state_mng.securityState = 0;

  return STATE_OK;
}
```

Listing 5.3: Listing Program Fungsi Set Current File

5.3 Get Current File

Berfungsi Sebagai getter untuk state member current file. Gambar 5.6 menampilkan DFD dari fungsi Get Current File. Diagram alir fungsi kemudian ditampilkan pada Gambar 5.7.

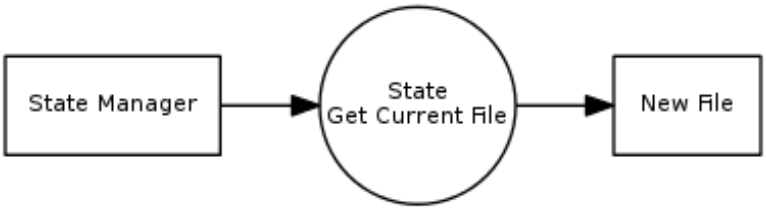

Figure 5.6: DFD Get Current File

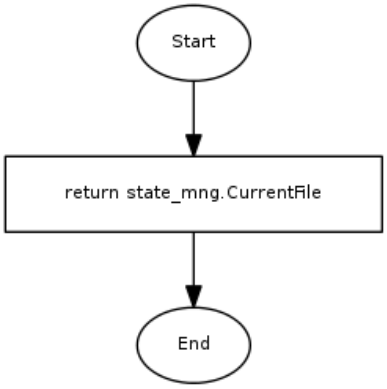

Figure 5.7: Flowchart Get Current File

5.3.1 Pengujian

Output
Return Value
Initialize state.currentFile to 0x0000
= 0x0000
Initialize state.currentFile to 0x000f
= 0x000f
Initialize state.currentFile to 0x00ff
= 0x00ff
Initialize state.currentFile to 0x0fff
= 0x0fff
Initialize state.currentFile to 0xffff
= 0xffff

Table 5.6: Test Vector Fungsi State Get Current File

Tabel 5.6 menampilkan Test Vector yang digunakan untuk menguji fungsi State Get Current File.

5.3.2 Implementasi

Tabel 5.7 menampilkan purwarupa dari implementasi fungsi Get Current File.

Name	Get Current File
Input	new current file
Output	State Manager

Table 5.7: Prototype Fungsi Get Current File

Listing 5.4 menampilkan potongan program yang mengimplementasi fungsi Get Current File.

```
1 uint16_t State_GetCurrent()
2 {
3   return state_mng.current;
4 }
```

Listing 5.4: Listing Program Fungsi Get Current File

5.4 Get Security State

Berfungsi Sebagai getter untuk state member security state. Gambar 5.8 menampilkan DFD dari fungsi Get Security State. Diagram alir fungsi kemudian ditampilkan pada Gambar 5.9.

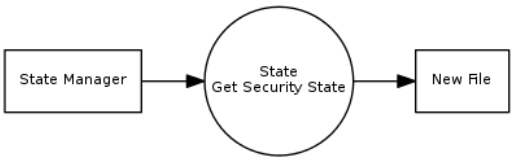

Figure 5.8: DFD Get Security State

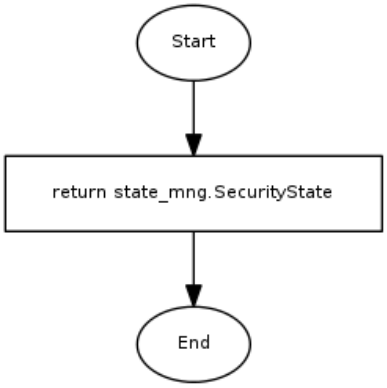

Figure 5.9: Flowchart Get Security State

5.4.1 Pengujian

Tabel 5.8 menampilkan Test Vector yang digunakan untuk menguji fungsi State Get Security State.

5.4.2 Implementasi

Tabel 5.9 menampilkan purwarupa dari implementasi fungsi State Get Security State.

Listing 5.5 menampilkan potongan program yang mengimplementasi fungsi Get Security State

```
1 uint8_t State_GetSecurityState()
2 {
```

Output
Return Value
Initialize state.securityState to 0x00
= 0x00
Initialize state.securityState to 0x01
= 0x01
Initialize state.securityState to 0x02
= 0x02
....
Initialize state.securityState to 0xff
= 0xff

Table 5.8: Test Vector Fungsi State Get Security State

Name	State_GetSecurityState
Input	-
Output	SecurityState

Table 5.9: Prototype Fungsi Get Security State

```
3   return state_mng.securityState;
4 }
```

Listing 5.5: Listing Program Fungsi Get Security State

5.5 Get Challenge

Berfungsi menghasilkan bilangan acak (*challange*) yang digunakan untuk otentikasi eksternal. Gambar 5.10 menampilkan DFD dari fungsi Get Challenge. Diagram alir fungsi kemudian ditampilkan pada Gambar 5.11.

5.5.1 Pengujian

Tabel 5.10 menampilkan Test Vector yang digunakan untuk menguji fungsi State Get Challenge.

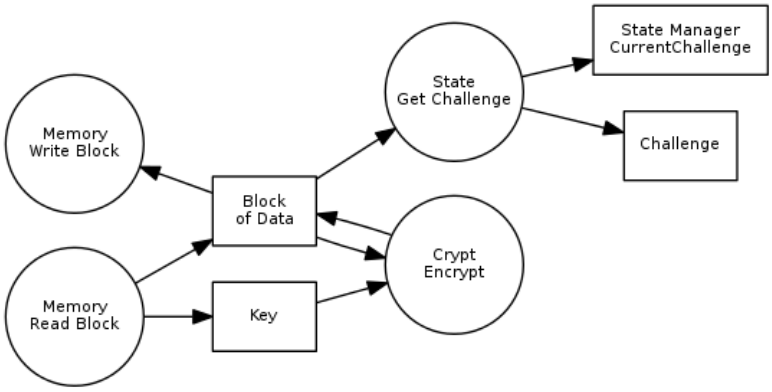

Figure 5.10: DFD Get Challenge

Output	
challenge	**(New) Key**
Initialize data block (Memory) to 0xxxxxx Initialize key (Memory) to 0xxxxx	
=0xxxxx	=0xxxx

Table 5.10: Test Vector Fungsi State Get Challenge

5.5.2 Implementasi

Tabel 5.11 menampilkan purwarupa dari implementasi fungsi Get Challenge.

Name	State_GetChallenge
Input	
Output	pointer ke buffer challenge

Table 5.11: Prototype Fungsi Get Challenge

Listing 5.6 menampilkan potongan program yang mengimplementasi fungsi Get Challenge.

```
1 void State_GetChallenge( uint8_t * buffer )
2 {
3   iu32 block[2], key[4];
4
5   HAL_Mem_ReadBlock( (uint16_t) SERNUM_ADDR, (uint16_t) SERNUM_LEN,
        (uint8_t *) block);
```

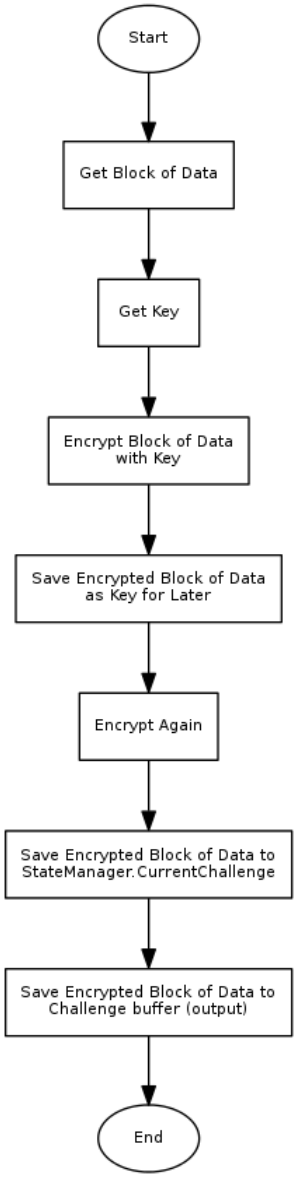

Figure 5.11: Flowchart Get Challenge

```
6  /* hal_eeprom_read( (iu8*)block, SERNUM_ADDR, SERNUM_LEN ); */
7
8  HAL_Mem_ReadBlock( (uint16_t) RAND_STATE_ADDR, (uint16_t)
       sizeof(key), (uint8_t *) key);
9  /* hal_eeprom_read( (iu8*)key, RAND_STATE_ADDR, sizeof(key) ) */
10
11 key[2]=key[1];
```

```
12   key[3]=key[0];
13
14   crypt_enc( block, key );
15
16   HAL_Mem_WriteBlock( (uint16_t) RAND_STATE_ADDR, (uint16_t)
         RAND_STATE_LEN, (uint8_t *) block);
17
18   crypt_enc( block, key );
19
20   memcpy( state_mng.challenge, block, sizeof(block) );
21
22   memcpy( buffer, block, CRYPT_BLOCK_LEN );
23 }
```

Listing 5.6: Listing Program Fungsi Get Challenge

5.6 Verify (PIN)

Berfungsi untuk otentikasi dasar menggunakan PIN. Gambar 5.12 menampilkan DFD dari fungsi State Verify. Diagram alir fungsi kemudian ditampilkan pada Gambar 5.13.

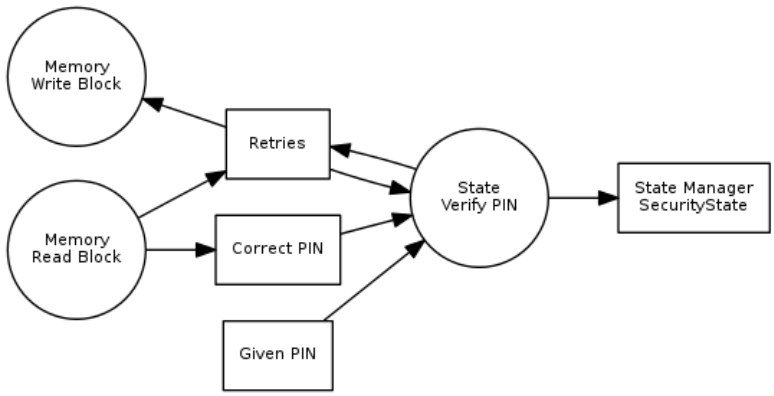

Figure 5.12: DFD Verify PIN

5.6.1 Pengujian

Tabel 5.12 menampilkan Test Vector yang digunakan untuk menguji fungsi State Verify (PIN).

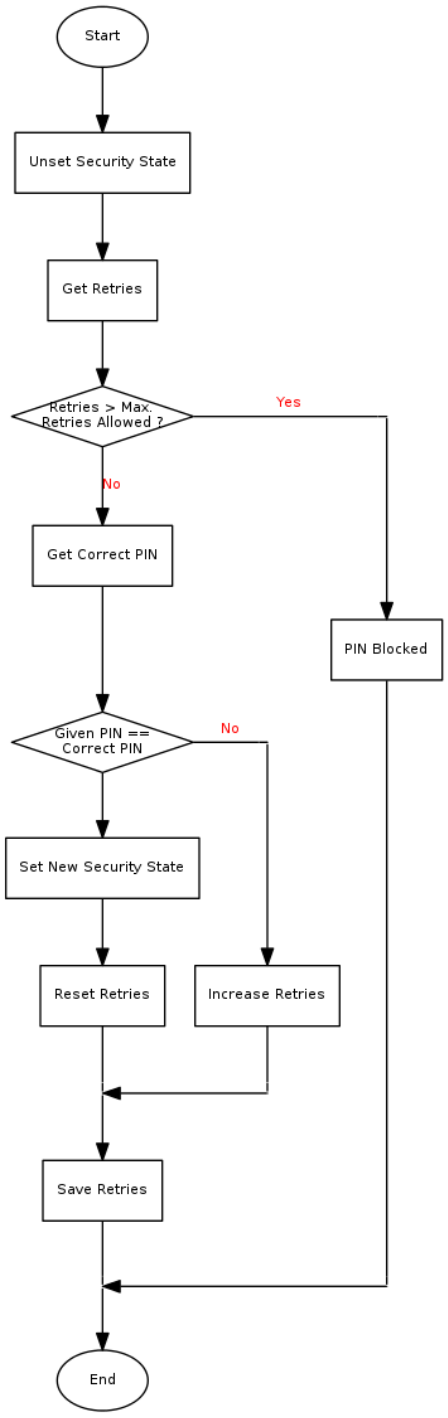

Figure 5.13: Flowchart Verify PIN

Input	Output		
Given PIN	**State**	**Retries**	**Ret. Value**
Initialize PIN (Memory) to 0x1234 Initialize Retries (Memory) to 0 Initialize SecurityState (State) to 0			
0x1234	=1	=0	=STATE_OK
0x1111	=0	=1	=STATE_WRONG
0x1111	=0	=2	=STATE_WRONG
0x1111	=0	=3	=STATE_WRONG
0x1111	=0	=3	=STATE_BLOCK

Table 5.12: Test Vector Fungsi State Verify

5.6.2 Implementasi

Tabel 5.13 menampilkan purwarupa dari implementasi fungsi Verify PIN.

Name	State_Verify
Input	PIN
Output	Result Status

Table 5.13: Prototype Fungsi Verify

Listing 5.7 menampilkan potongan program yang mengimplementasi fungsi Verify.

```
1  int State_Verify(uint8_t *pin)
2  {
3    state_mng.securityState &= 0xfe;
4
5    uint8_t retries;
6
7    retries = HAL_Mem_ReadByte(PIN_RETRIES_ADDR);
8
9    if (retries > PIN_MAX_RETRIES)
10     {
11       return STATE_BLOCKED;
12     }
13
14   uint8_t i, temp, diff=0;
```

```
15   for( i=0; i<PIN_LEN; i++ )
16     {
17       temp = HAL_Mem_ReadByte(PIN_ADDR+i);
18       /* HAL_IO_TxByte(pin[i]); */
19       /* HAL_IO_TxByte(temp); */
20       diff |= temp^pin[i];
21     }
22
23   if( diff>0 )
24     {
25       retries++;
26     }
27   else
28     {
29       retries=0;
30     }
31
32   HAL_Mem_WriteByte(PIN_RETRIES_ADDR, retries);
33
34   if( diff>0 )
35     {
36       return STATE_WRONG;
37     }
38
39   state_mng.securityState |= 0x01;
40
41   return STATE_OK;
42 }
```

Listing 5.7: Listing Program Fungsi Verify

5.7 Verify Auth

Berfungsi untuk verifikasi user lanjutan menggunakan metode otentikasi. Gambar 5.14 menampilkan DFD dari fungsi State Verify Key. Diagram alir fungsi kemudian ditampilkan pada Gambar 5.15.

5.7.1 Pengujian

Tabel 5.14 menampilkan Test Vector yang digunakan untuk menguji fungsi State Verify Auth.

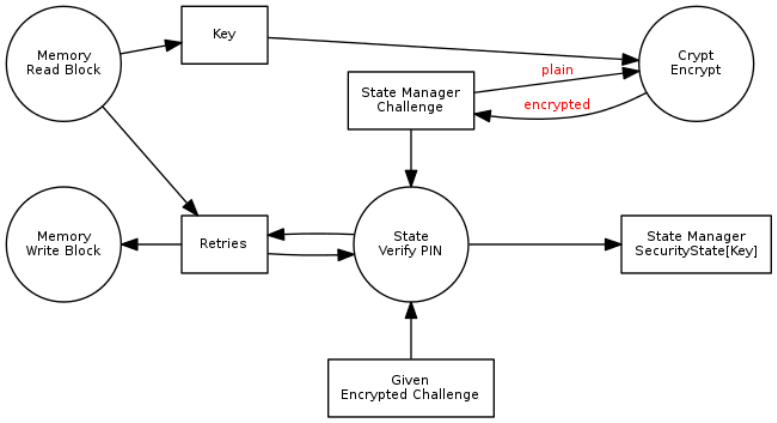

Figure 5.14: DFD Verify Auth

Input	Output		
Given Enc. Challenge	**State**	**Retries**	**Ret. Value**
Initialize Key (Memory) to 0xxxxx Initialize Retries (Memory) to 0 Initialize Challenge (State) to 0 Initialize SecurityState (State) to 0			
=0xxxxx	=1	=0	=STATE_OK
=0xxxxx	=0	=1	=STATE_WRONG
=0xxxxx	=0	=2	=STATE_WRONG
=0xxxxx	=0	=3	=STATE_WRONG
=0xxxxx	=0	=3	=STATE_BLOCK

Table 5.14: Test Vector Fungsi State Verify Auth

5.7.2 Implementasi

Tabel 5.15 menampilkan purwarupa dari implementasi fungsi Verify Auth.

Listing 5.8 menampilkan potongan program yang mengimplementasi fungsi Verify Auth

```
1 uint8_t State_VerifyAuth( uint8_t * encrypted )
2 {
3   uint8_t retries;
4   retries = HAL_Mem_ReadByte(EXT_AUTH_RETRIES_ADDR);
5
6   if (retries > EXT_AUTH_MAX_RETRIES)
7     {
8       return STATE_BLOCKED;
```

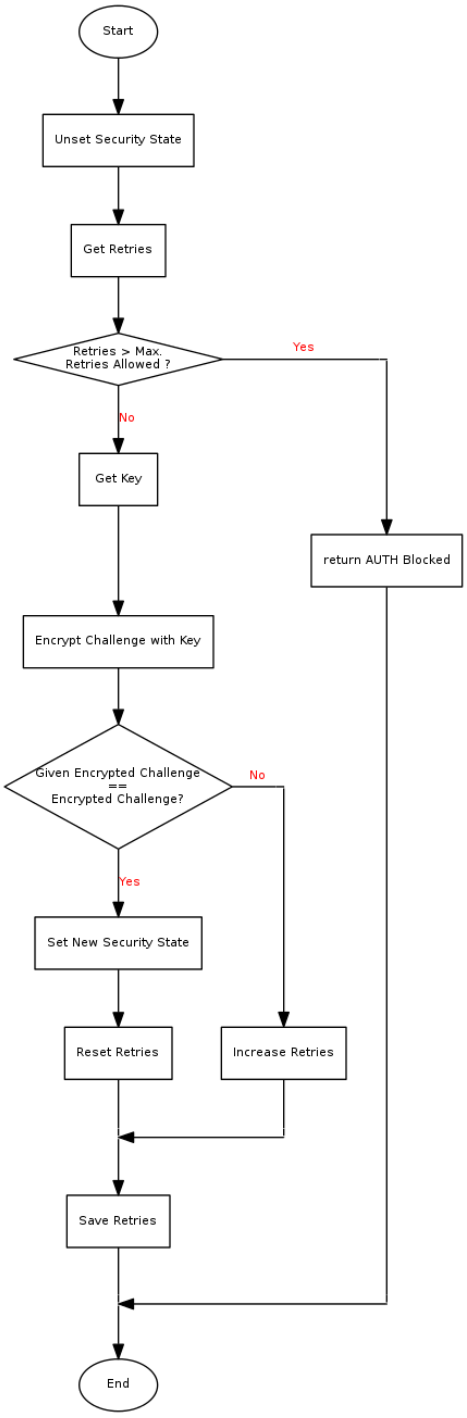

Figure 5.15: Flowchart Verify Auth

Name	State_VerifyAuth
Input	Encrypted Challenge
Output	Result Status

Table 5.15: Prototype Fungsi Verify Auth

```
9     }
10
11    uint32_t key[4];
12    HAL_Mem_ReadBlock( (uint16_t) EXT_AUTH_KEY_ADDR, (uint16_t)
          EXT_AUTH_KEY_LEN, (uint8_t *) key);
13
14    crypt_enc( state_mng.challenge, key );
15
16    /* Compare result */
17    if( memcmp( encrypted, state_mng.challenge, CRYPT_BLOCK_LEN ) ) {
18      retries++;
19      HAL_Mem_WriteByte(EXT_AUTH_RETRIES_ADDR, retries);
20
21      return STATE_WRONG;
22    }
23
24    if(retries > 0)
25      {
26        retries = 0;
27        HAL_Mem_WriteByte(EXT_AUTH_RETRIES_ADDR, retries);
28      }
29
30    state_mng.securityState |= 0x02;
31
32    return STATE_OK;
33  }
```

Listing 5.8: Listing Program Fungsi Verify Auth

Chapter 6

Command Handler

Command Handlers berfungsi untuk melaksanakan Instruksi yang terkandung didalam APDU Command. Terdapat satu dan hanya satu command handler untuk setiap instruksi. Tabel 6.1 menampilkan daftar instruksi yang didukung oleh pintarOS beserta kode instruksi yang sesuai. Untuk memudahkan penggunaan kode instruksi ini, dibuat kode makro untuk mendefiniskan kode untuk setiap instruksi.

Instruksi	Kode
SELECT	A4
READ BINARY	B0
UPDATE BINARY	D6
READ RECORD	B2
UPDATE RECORD	DC
APPEND RECORD	E2
CREATE FILE	E0
DELETE FILE	E4
VERIFY (PIN)	20
EXTERNAL AUTH	82
GET CHALLENGE	84
GET RESPONSE	C0

Table 6.1: Daftar Instruksi beserta kode yang digunakan

Setiap instruksi akan dilaksanakan oleh command handler yang sesuai, sehingga diperlukan satu fungsi lainnya yang akan menentukan command handler mana yang akan dipanggil berdasarkan kode instruksi yang terdapat pada command APDU.

Setiap command handler bekerja berdasarkan command APDU, yang disimpan dalam sebuah array berukuran 5 dengan tipe data unsigned integer

8 bit. Listing 6.1 menampilkan deklarasi array header sebagaimana dapat ditemukan pada file header *command.h*.

```
1 uint8_t header[5];
```

Listing 6.1: Deklarasi array header

Selain malaksanakan instruksi berdasarkan command APDU header, setiap command handler juga bertanggung jawab memberikan Response Type yang sesuai kepada Response Manager untuk selanjutnya diubah menjadi status word dari pesan Response APDU.

Tabel 6.2 menampilkan fungsi-fungsi yang terdapat pada bagian Command Handlers beserta kegunaannya.

Nama Fungsi	**Kegunaan**
Command Interpreter	Memanggil Command Handler yang sesuai berdasarkan instruksi
Command Select	Command Handler untuk Instruksi Select
Command Read Binary	Command Handler untuk Instruksi Read Binary
Command Update Binary	Command Handler untuk Instruksi Update Binary
command Read Record	Command Handler untuk Instruksi Read Record
Command Update Record	Command Handler untuk Instruksi Update Record
Command Append Record	Command Handler untuk Instruksi Append Record
Command Create File	Command Handler untuk Instruksi Create File
Command Delete File	Command Handler untuk Instruksi Delete File
Command Verify (pin)	Command Handler untuk Instruksi Verify
Command External Auth	Command Handler untuk Instruksi External Auth
Command Get Challenge	Command Handler untuk Instruksi Get Challenge
Command Get Response	Command Handler untuk Instruksi Get Response

Table 6.2: Daftar Response Type dan Status Word yang sesuai

6.1 Command Interpreter

Berfungsi menemukan dan memanggil command handler yang sesuai dengan Instruksi pada Command APDU. Gambar 6.1 menampilkan DFD dari fungsi Command Interpreter. Diagram alir fungsi kemudian ditampilkan pada Gambar 6.2.

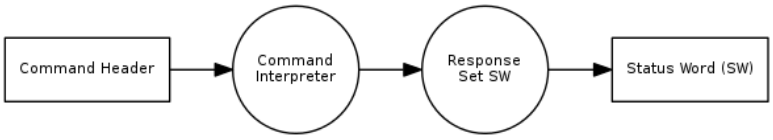

Figure 6.1: DFD Command Interpreter

6.1.1 Pengujian

Input	**Output**
Header	**Status Word**
[0x00,0x00,0x00.0x00,0x00]	= 6E00 (CLA Not Supported)
[0x80,0x00,0x00,0x00,0x00]	= 6D00 (INS Not Supported)
Include all test from Command Select	
Include all test from Read Binary	
Include all test from Update Binary	
Include all test from Create File	
Include all test from Delete File	
Include all test from Verify	
Include all test from Get Challenge	
Include all test from External Auth	
Include all test from Get Response	

Table 6.3: Test Vector Fungsi Command Interpreter

Tabel 6.3 menampilkan Test Vector yang digunakan untuk menguji fungsi Command Interpreter.

6.1.2 Implementasi

Tabel 6.4 menampilkan purwarupa dari implementasi fungsi Command Interpreter.

Listing 6.2 menampilkan kode program yang mengimplementasi fungsi Command Interpreter dan dapat ditemukan pada file *command.c*.

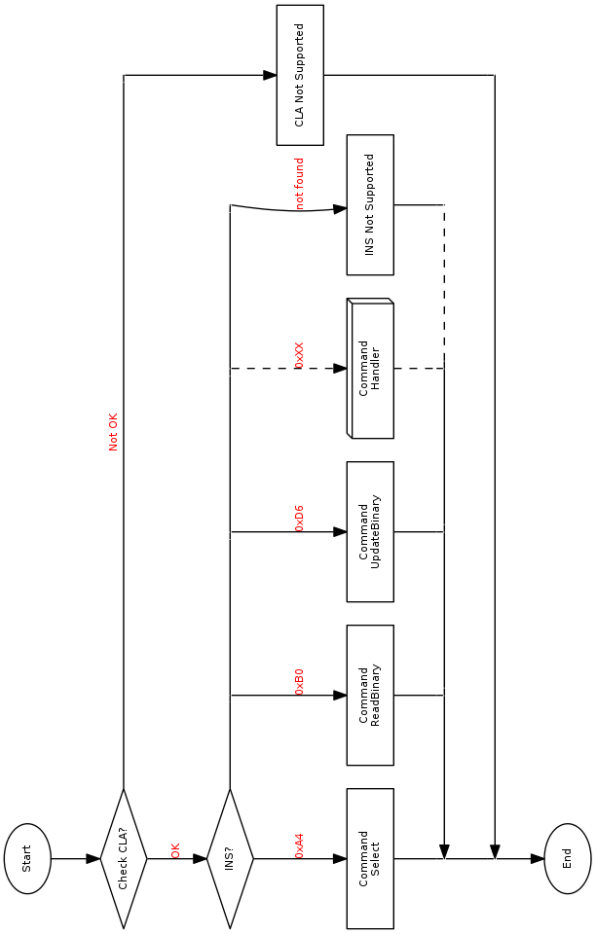

Figure 6.2: Flowchart Command Interpreter

```
void Command_Interpreter()
{
      if ( (header[0]&0xFC)==0x80 )
        {
          switch( header[1]&0xFE )
            {
            case ISO_SELECT:
              Command_Select();
              break;
            case ISO_READ_BINARY:
              Command_ReadBinary();
              break;
            case ISO_UPDATE_BINARY:
              Command_UpdateBinary();
```

Name	Command_Interpreter
Input	-
Output	-

Table 6.4: Prototype Fungsi Command Interpreter

```
15              break;
16            case ISO_CREATE_FILE:
17              Command_CreateFile();
18              break;
19            case ISO_DELETE_FILE:
20              Command_DeleteFile();
21              break;
22            case ISO_VERIFY:
23              Command_Verify();
24              break;
25            case ISO_GET_CHALLENGE:
26              Command_GetChallenge();
27              break;
28            case ISO_EXT_AUTH:
29              Command_ExternalAuth();
30              break;
31            case ISO_GET_RESPONSE:
32              Command_GetResponse();
33              break;
34            default:
35              Response_SetSW( Response_INSNotSupported, 0 );
36              break;
37            }
38        }
39      else
40        {
41          Response_SetSW( Response_CLANotSupported, 0);
42        }
43  }
```

Listing 6.2: Implementasi Fungsi Command Interpreter

6.2 Command Handler - Select

Berfungsi sebagai command handler untuk instruksi Select. Gambar 6.3 menampilkan DFD dari fungsi Command Handler Select. Diagram alir fungsi kemudian ditampilkan pada Gambar 6.4.

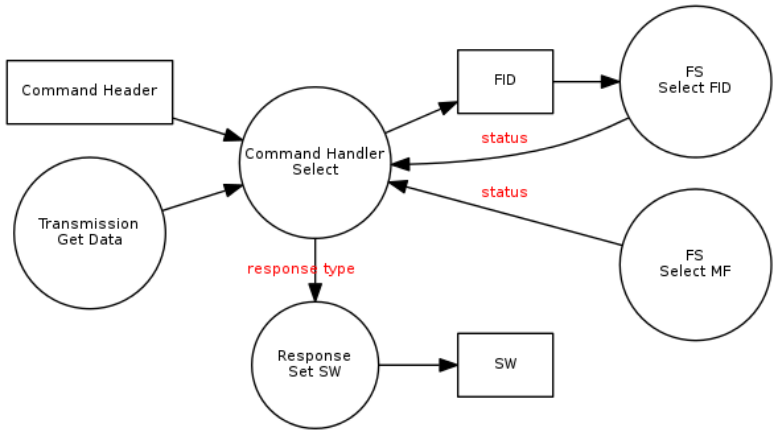

Figure 6.3: DFD Command Handler Select

6.2.1 Pengujian

Input		Output
Header	**Data**	**Status Word**
[0x80,0xA4,0x00.0x00,0x00]	-	= 9000 (OK)
[0x80,0xA4,0x00,0x00,0x02]	[3F,00]	= 9000 (OK)
[0x80,0xA4,0x00,0x00,0x02]	[FF,FF]	= 6A82 (File Not Found)
[0x80,0xA4,0xFF,0xFF,0x00]	-	= 6A00 (Wrong P1P2)

Table 6.5: Test Vector Fungsi Command Handler Select

Tabel 6.5 menampilkan Test Vector yang digunakan untuk menguji fungsi Command Select.

6.2.2 Implementasi

Tabel 6.6 menampilkan purwarupa dari implementasi fungsi Command Select. Listing 6.3 menampilkan potongan program yang mengimplementasi fungsi Command Handler Select

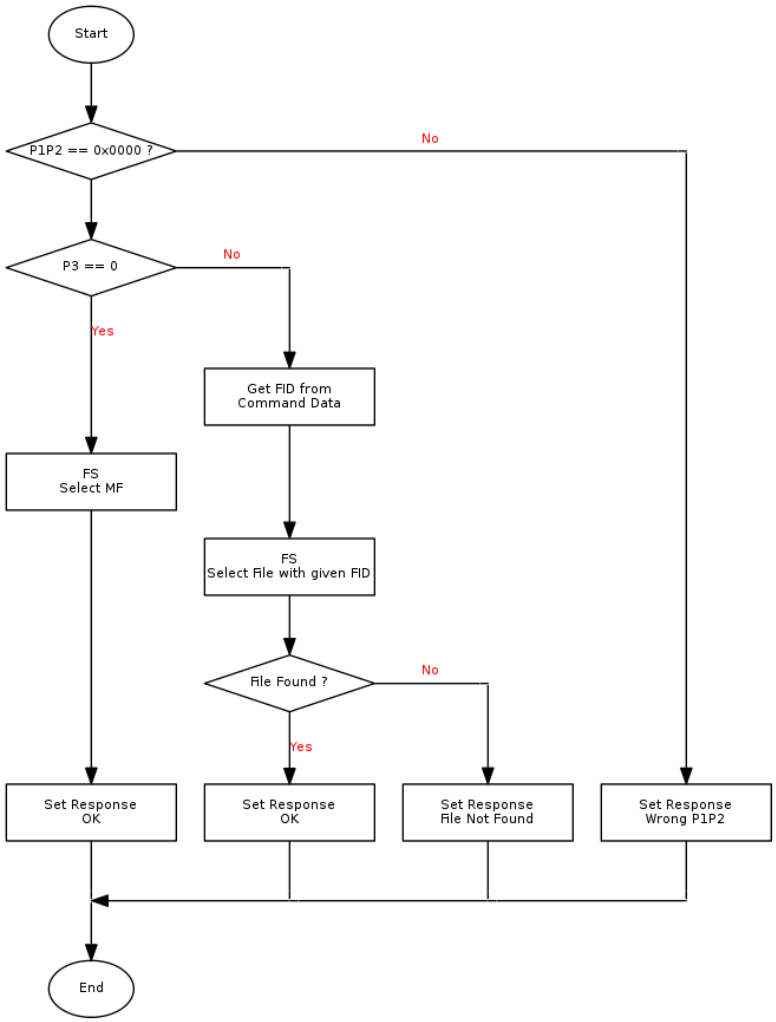

Figure 6.4: Flowchart Command Handler Select

Name	Command_Select
Input	-
Output	-

Table 6.6: Prototype Fungsi Command Handler - Select

```
void Command_Select()
{
  uint16_t fid, file;
  uint8_t data, temp;

  if((header[2] == 0x00) && (header[3] == 0x00))
    {
      /* Select MF */
      if(header[4] == 0)
        {
          FSSelectMF();
          Response_SetSW(Response_OK, 0);
        }
      else
        {
          /* Get FID */
          Transmission_GetData(&temp,1);
          fid = (uint6_t) (temp) << 8;
          Transmission_GetData((&temp,1);
          fid |= (uint16_t) temp;

          /* Select with FID */
          switch( FS_SelectFID(fid) )
            {
            case FS_OK:
              Response_SetSW(Response_OK, 0);
              break;
            case FS_ERROR_NOT_FOUND:
              Response_SetSW( Response_WrongP1P2_FileNotFound, 0);
              break;
            }
        }
    }
```

```
34 }
```

Listing 6.3: Listing Program Fungsi Command Handler Select

6.3 Command Handler - Read Binary

Berfungsi sebagai command handler untuk instruksi Read Binary. Gambar 6.5 menampilkan DFD dari fungsi Command Handler Read Binary. Diagram alir fungsi kemudian ditampilkan pada Gambar 6.6.

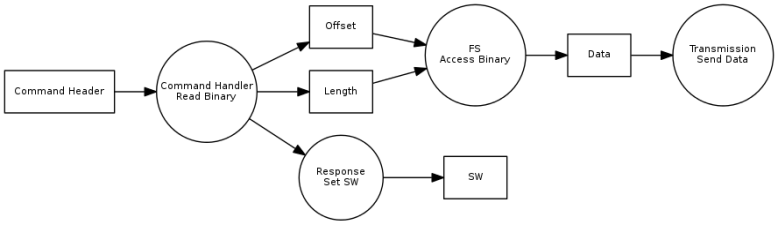

Figure 6.5: DFD Command Read Binary

6.3.1 Pengujian

Input	**Output**	
Header	**Data**	**Status Word**
Create File with AC Read = 0 Select the last created file Update File with data [1,2,3,4,5]		
[0x80,0xB0,0x00.0x00,0x05]	= [1,2,3,4,5]	= 6100 (Normal)
Create File with AC Read $\geq$ 1 Select the last created file Update File with data [1,2,3,4,5]		
[0x80,0xB0,0x00,0x00,0x05]	-	= 6982 (Not Allowed)

Table 6.7: Test Vector Fungsi Command Handler Read Binary

Tabel 6.7 menampilkan Test Vector yang digunakan untuk menguji fungsi Command Read Binary.

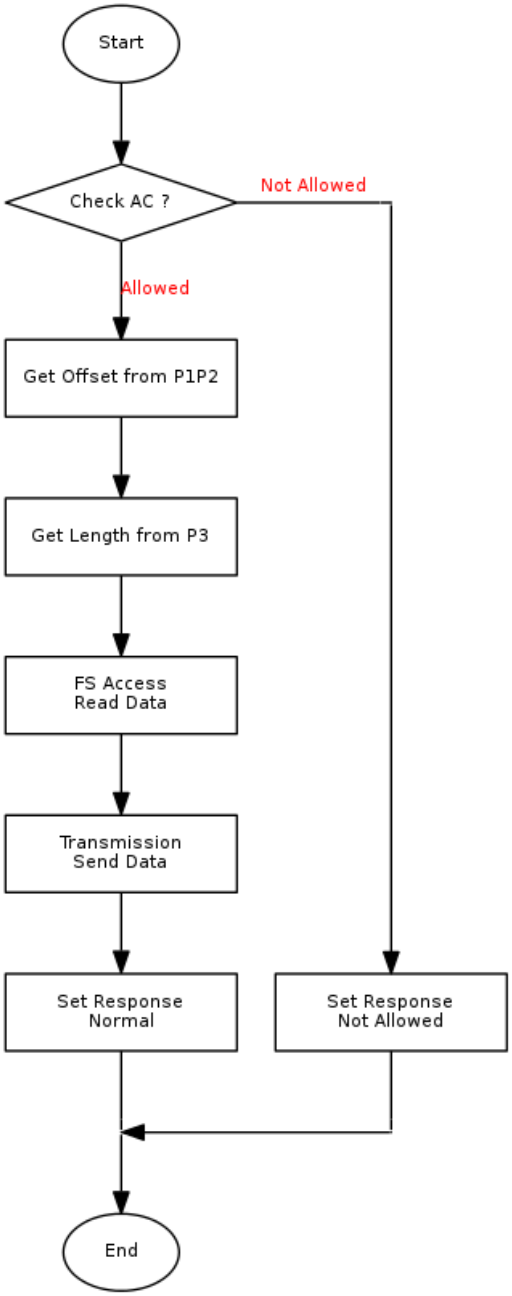

Figure 6.6: Flowchart Command Handler Read Binary

6.3.2 Implementasi

Tabel 6.8 menampilkan purwarupa dari implementasi fungsi Command Read Binary.

Name	Command_ReadBinary
Input	-
Output	-

Table 6.8: Prototype Fungsi Command Handler Read Binary

Listing 6.4 menampilkan potongan program yang mengimplementasi fungsi Command Handler Read Binary

```
1  void Command_ReadBinary()
2  {
3    uint16_t offset, length;
4    uint8_t data;
5
6    HAL_IO_TxByte( header[1] );
7
8    if( !(FS_CheckAC(FS_OP_READ) == FS_OK))
9      {
10       Response_SetSW( Response_CmdNotAllowed_SecurityStatus , 0);
11       return;
12     }
13
14   offset = ((uint16_t)header[2] << 8) | header[3];
15
16   uint8_t i;
17   for( i=0; i<header[4]; i++ )
18     {
19       FSAccessBinary(FS_OP_READ,offset+i,1,&data);
20       Transmission_SendData(&data,1);
21     }
22
23   Response_SetSW( Response_Normal, header[4]-i );
24 }
```

Listing 6.4: Kode Progrem Fungsi Command Handler Read Binary

6.4 Command Handler - Update

Berfungsi sebagai command handler untuk instruksi Update Binary. Gambar 6.7 menampilkan DFD dari fungsi Command Handler Update. Diagram alir fungsi kemudian ditampilkan pada Gambar 6.8.

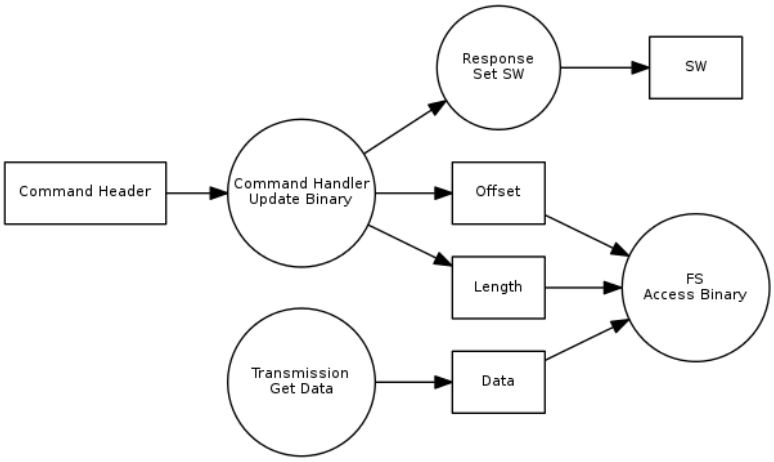

Figure 6.7: DFD Command Update

6.4.1 Pengujian

Input		Output
Header	**Data**	**Status Word**
Create File with AC Update = 0 Select the last created file		
[0x80,0xB0,0x00.0x00,0x05]	[1,2,3,4,5]	= 6100 (Normal)
Create File with AC Read $\geq$ 1 Select the last created file		
[0x80,0xD6,0x00,0x00,0x05]	[1,2,3,4,5]	= 6982 (Not Allowed)

Table 6.9: Test Vector Fungsi Command Handler Update Binary

Tabel 6.7 menampilkan Test Vector yang digunakan untuk menguji fungsi Command Read Binary.

6.4.2 Implementasi

Tabel 6.10 menampilkan purwarupa dari implementasi fungsi Command Update Binary.

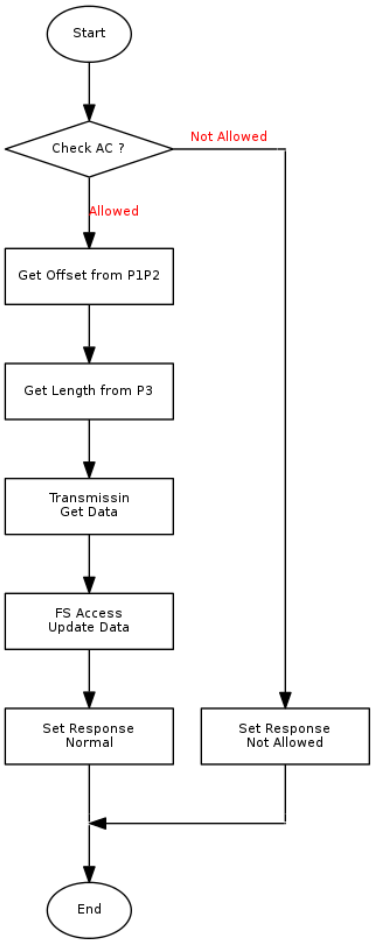

Figure 6.8: Flowchart Command Handler Update

Listing 6.5 menampilkan potongan program yang mengimplementasi fungsi Command Handler Update Binary.

```
1  void Command_UpdateBinary()
2  {
3    uint16_t offset, length;
4    uint8_t data;
5
6    if( !(FS_CheckAC(FS_OP_UPDATE) == FS_OK))
7      {
8        Response_SetSW( Response_CmdNotAllowed_SecurityStatus , 0);
9        return;
10     }
11
```

Name	Command_UpdateBinary
Input	-
Output	-

Table 6.10: Prototype Fungsi Command Handler Update

```
12   offset = ((uint16_t)(header[2]) << 8) | header[3];
13   length = (uint16_t) header[4];
14
15   uint8_t i;
16   for( i=0; i<length; i++)
17     {
18       Transmission_GetData(&data,1);
19       FSAccessBinary(FS_OP_UPDATE,offset+i,1,&data);
20     }
21
22   Response_SetSW( Response_Normal , 0);
23 }
```

Listing 6.5: Kode Program Fungsi Command Handler Update

6.5 Command Handler - Create File

Berfungsi sebagai command handler untuk instruksi Create File. Gambar 6.9 menampilkan DFD dari fungsi Command Handler Create File. Diagram alir fungsi kemudian ditampilkan pada Gambar 6.10.

6.5.1 Pengujian

Tabel 6.7 menampilkan Test Vector yang digunakan untuk menguji fungsi Command Read Binary.

6.5.2 Implementasi

Tabel 6.12 menampilkan purwarupa dari implementasi fungsi Command Create File.

Listing **??** menampilkan potongan program yang mengimplementasi fungsi Command Handler Create File

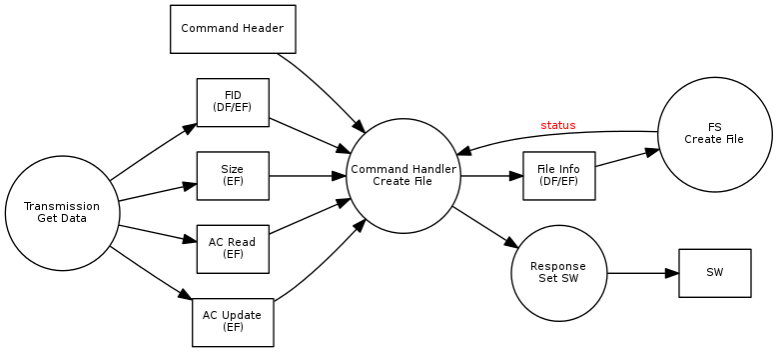

Figure 6.9: DFD Command Create File

Input		Output
Header	**Data**	**Status Word**
Create DF File		
[0x80,0xE0,0x00.0x00,0x03]	[0x4f,0x12,0x34]	= 6100 (Normal)
Create DF with FID 3F00		
[0x80,0xE0,0x00.0x00,0x03]	[0x4f,0x3f,0x00]	= 6F00 (Fatal Error)
Create EF with FID 5678		
[0x80,0xE0,0x00,0x00,0x07]	[0x5f,0x56,0x78,0x00,0x05,0x00,0x00]	= 6100 (Normal)
Create EF with FID 5678 (same with the last created		
[0x80,0xE0,0x00,0x00,0x07]	[0x5f,0x56,0x78,0x00,0x05,0x00,0x00]	= 6F00 (Fatal Error)

Table 6.11: Test Vector Fungsi Command Handler Update Binary

```
1  void Command_CreateFile()
2  {
3    uint8_t tag;
4    uint16_t size, fid;
5    uint8_t acread, acupdate;
6    uint8_t result;
7    uint8_t temp;
8
9    /* Get Tag */
10   Transmissin_GetData(&tag,1)
11
12   switch( tag )
13     {
14     case FS_TAG_DF:
15       /* Get FID */
16       Transmission_GetData(&temp,1);
17       fid = (uint6_t) (temp) << 8;
```

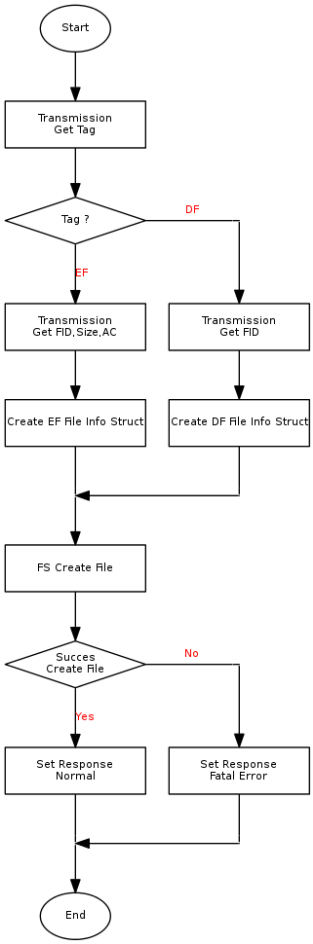

Figure 6.10: Flowchart Command Handler Create File

```
18      Transmission_GetData((&temp,1);
19      fid |= (uint16_t) temp;
20
21      struct DF_st df;
22      df.FID = fid;
23
24      result = FSCreateFile(tag,&df);
25      break;
26    case FS_TAG_EF:
27      /* Get FID */
28      Transmission_GetData(&temp,1);
29      fid = (uint6_t) (temp) << 8;
30      Transmission_GetData((&temp,1);
31      fid |= (uint16_t) temp;
```

Name	Command_CreateFile
Input	-
Output	-

Table 6.12: Prototype Fungsi Command Handler Create File

```
32
33      /* Get Size */
34      Transmission_GetData(&temp,1);
35      size = (uint6_t) (temp) << 8;
36      Transmission_GetData((&temp,1);
37      size |= (uint16_t) temp;
38
39      /* Get AC for Read */
40      Transmission_GetData(&acread,1);
41
42      /* Get AC for Update */
43      Transmission_GetData(&acupdate,1);
44
45      struct EF_st ef;
46
47      ef.FID = fid;
48      ef.structure = FS_EF_STRUCTURE_TRANSPARENT;
49      ef.type = FS_EF_TYPE_WORKING;
50      ef.ACRead = acread; //(ac && 0xf0) >> 4;
51      ef.ACUpdate = acupdate; //ac && 0x0f;
52      ef.size = size;
53
54      result = FSCreateFile(tag,&ef);
55
56      break;
57    }
58
59  switch ( result )
60    {
61    case FS_OK:
62      Response_SetSW( Response_Normal , 0);
63      break;
64    default:
65      Response_SetSW( Response_FatalError , 0);
```

```
66        break;
67      }
68  }
```

Listing 6.6: Listing Program Fungsi Command Handler Create File

6.6 Command Handler - Delete File

Berfungsi sebagai command handler untuk instruksi Delete File. Gambar 6.11 menampilkan DFD dari fungsi Command Handler Delete File. Diagram alir fungsi kemudian ditampilkan pada Gambar 6.12.

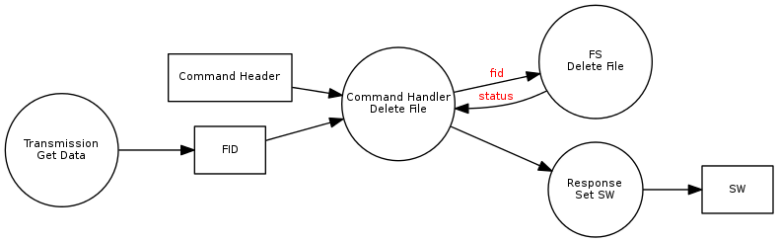

Figure 6.11: DFD Command Delete File

6.6.1 Pengujian

Input		**Output**
Header	**Data**	**Status Word**
Description : *Delete Existence File* **Setup :** *Create File with FID 1234*		
[0x80,0xE4,0x00.0x00,0x02]	[0x12,0x34]	= 6100 (Normal)
Description : *Delete Non-existence File*		
[0x80,0xE4,0x00.0x00,0x03]	[0x56,0x78]	= 6F00 (Fatal Error)

Table 6.13: Test Vector Fungsi Command Handler Delete File

Tabel 6.13 menampilkan Test Vector yang digunakan untuk menguji fungsi Command Handler Delete File.

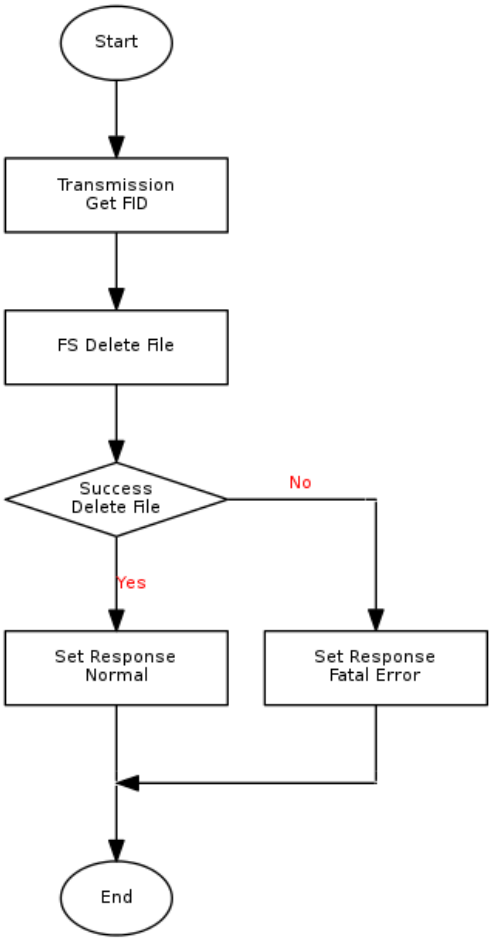

Figure 6.12: Flowchart Command Handler Delete File

6.6.2 Implementasi

Tabel 6.14 menampilkan purwarupa dari implementasi fungsi Command Delete File.

Listing 6.7 menampilkan kode program yang mengimplementasi fungsi Command Handler Delete File

```
1 void Command_DeleteFile()
2 {
3   uint16_t fid;
4   uint8_t temp;
5
6   /* Get FID */
7   Transmission_GetData(&temp,1);
```

Name	Command_DeleteFile
Input	-
Output	-

Table 6.14: Prototype Fungsi Command Handler Delete File

```
8   fid = (uint6_t) (temp) << 8;
9   Transmission_GetData((&temp,1);
10  fid |= (uint16_t) temp;
11
12  switch ( FSDeleteFile(fid) )
13    {
14    case FS_OK:
15      Response_SetSW( Response_Normal , 0);
16      break;
17    default:
18      Response_SetSW( Response_FatalError , 0);
19      break;
20    }
21 }
```

Listing 6.7: Kode Program Fungsi Command Handler Delete File

6.7 Command Handler - Verify PIN

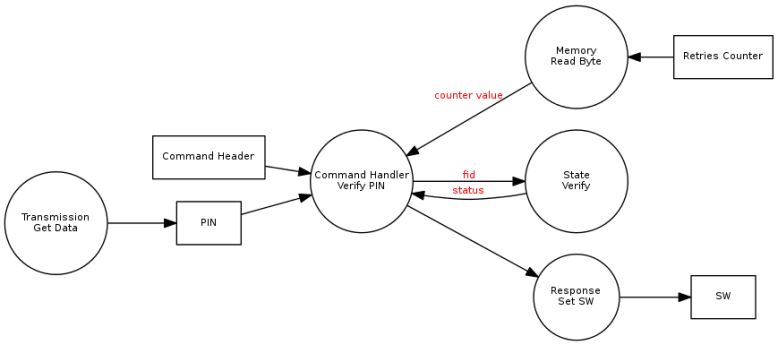

Figure 6.13: DFD Command Handler Verify

Berfungsi sebagai command handler untuk instruksi Verify. Gambar 6.13

menampilkan DFD dari fungsi Command Handler Verify. Diagram alir fungsi kemudian ditampilkan pada Gambar 6.14.

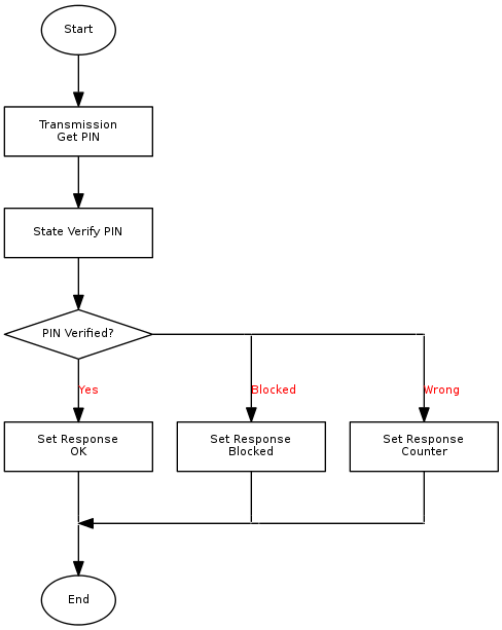

Figure 6.14: Flowchart Command Handler Verify

6.7.1 Pengujian

Tabel 6.15 menampilkan Test Vector yang digunakan untuk menguji fungsi Command Handler Verify.

6.7.2 Implementasi

Tabel 6.16 menampilkan purwarupa dari implementasi fungsi Command Handler Verify.

Listing 6.8 menampilkan potongan program yang mengimplementasi fungsi Command Handler Verify.

```
1 void Command_Verify()
2 {
3   uint8_t pin[PIN_LEN];
4
5   if( header[4] != PIN_LEN )
6     {
7       Response_SetSW( Response_WrongLength , 0);
```

Input		Output
Header	**Data**	**Status Word**
Description : *Verify with Correct PIN* **Setup :** *Correct PIN = 1234*		
[0x80,0x20,0x00.0x00,0x04]	[1,2,3,4]	= 9000 (OK)
Description : *Verify with Incorrect PIN until blocked* **Setup :** *Correct PIN = 1234* *Max. Try = 3*		
[0x80,0x20,0x00.0x00,0x04]	[4,3,2,1]	= 63C1 (Counter)
[0x80,0x20,0x00.0x00,0x04]	[4,3,2,1]	= 63C2 (Counter)
[0x80,0x20,0x00.0x00,0x04]	[4,3,2,1]	= 63C3 (Counter)
[0x80,0x20,0x00.0x00,0x04]	[4,3,2,1]	= 6383 (Auth Blocked)

Table 6.15: Test Vector Fungsi Command Handler Verify

Name	Command_Verify
Input	-
Output	-

Table 6.16: Prototype Fungsi Command Handler Verify

```
8       return;
9     }
10
11  /* Get PIN */
12  Transmission_GetData(pin, PIN_LEN);
13
14  uint8_t retries;
15  switch( State_Verify(pin) )
16    {
17    case STATE_OK:
18      Response_SetSW( Response_OK , 0);
19      break;
20    case STATE_BLOCKED:
21      Response_SetSW( Response_CmdNotAllowed_AuthBlocked , 0);
```

```
22      break;
23    case STATE_WRONG:
24      retries = Memory_ReadByte(PIN_RETRIES_ADDR);
25      Response_SetSW( Response_Warning_Counter | (retries & 0x0f) ,
            0);
26    }
27  }
```

Listing 6.8: Listing Program Fungsi Command Handler Verify

6.8 Command Handler - Get Challenge

Berfungsi sebagai command handler untuk instruksi Get Challenge. Gambar 6.15 menampilkan DFD dari fungsi Command Handler Get Challenge. Diagram alir fungsi kemudian ditampilkan pada Gambar 6.16.

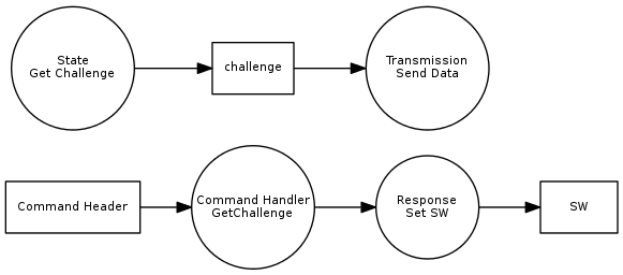

Figure 6.15: DFD Command Handler Get Challenge

6.8.1 Pengujian

Input	Output	
Header	**Data**	**Status Word**
[0x80,0x84,0x00.0x00,0x08]	x,x,x,x,x	= 6100 (Normal)

Table 6.17: Test Vector Fungsi Command Handler Get Challenge

Tabel 6.17 menampilkan Test Vector yang digunakan untuk menguji fungsi Command Handler Get Challenge.

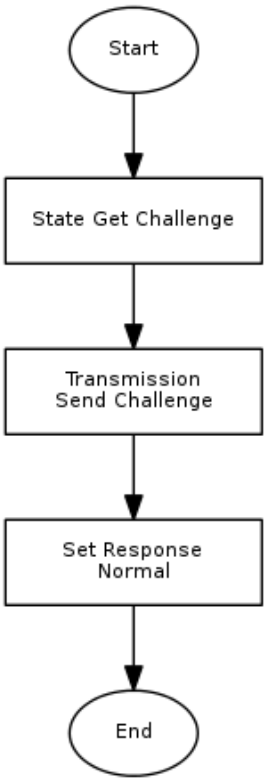

Figure 6.16: Flowchart Command Handler Get Challenge

6.8.2 Implementasi

Tabel 6.18 menampilkan purwarupa dari implementasi fungsi Command Handler Get Challenge.

Name	Command_GetChallenge
Input	-
Output	-

Table 6.18: Prototype Fungsi Command Handler Get Challenge

Listing 6.9 menampilkan potongan program yang mengimplementasi fungsi Command Handler Get Challenge.

```
1 void Command_GetChallenge()
2 {
```

```
3    uint8_t buf[CRYPT_BLOCK_LEN];
4
5    if( !(header[4] == CRYPT_BLOCK_LEN ) )
6          {
7            Response_SetSW( Response_WrongLength , 0);
8            return;
9          }
10
11   /* ACK */
12        HAL_IO_TxByte( header[1] );
13
14   State_GetChallenge( buf );
15
16        Transmission_SendData(buf, CRYPT_BLOCK_LEN);
17
18        Response_SetSW( Response_Normal , 0);
19 }
```

Listing 6.9: Listing Program Fungsi Command Handler Get Challenge

6.9 Command Handler - External Authentication

Berfungsi sebagai command handler untuk instruksi External Authentication. Gambar 6.17 menampilkan DFD dari fungsi Command Handler External Auth. Diagram alir fungsi kemudian ditampilkan pada Gambar 6.18.

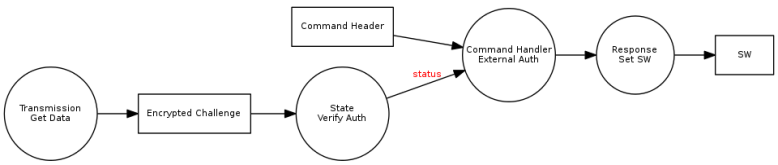

Figure 6.17: DFD Command External Auth

6.9.1 Pengujian

Tabel 6.19 menampilkan Test Vector yang digunakan untuk menguji fungsi Command Handler Verify.

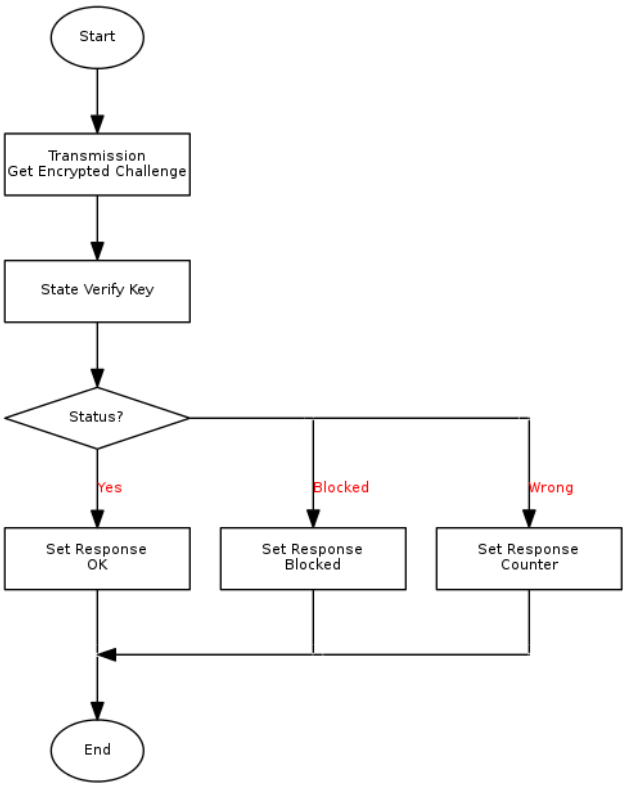

Figure 6.18: Flowchart Command Handler External Auth

6.9.2 Implementasi

Tabel 6.20 menampilkan purwarupa dari implementasi fungsi Command Handler External Authentication.

Listing 6.10 menampilkan potongan program yang mengimplementasi fungsi Command Handler External Authentication.

```
1  void Command_ExternalAuth()
2  {
3    uint8_t encrypted[CRYPT_BLOCK_LEN];
4    uint8_t i;
5
6    if( !(header[4] == CRYPT_BLOCK_LEN) )
7      {
8        Response_SetSW( Response_WrongLength , 0);
9        return;
10     }
11
```

Input		Output
Header	**Data**	**Status Word**
Description : *Verify with Correct Encrypted Challenge* **Setup :** *create challenge* *encrypted challenge = xxxxxxx*		
[0x80,0x82,0x00.0x00,0x08]	xxxxxxxx	= 9000 (OK)
Description : *Verify with Incorrect Encrypted Challenge* **Setup :** *encrypted challenge = xxxxxxx*		
[0x80,0x82,0x00.0x00,0x04]	xxxxxxxx	= 63C1 (Counter)
[0x80,0x82,0x00.0x00,0x04]	xxxxxxxx	= 63C2 (Counter)
[0x80,0x82,0x00.0x00,0x04]	xxxxxxxx	= 63C3 (Counter)
[0x80,0x82,0x00.0x00,0x04]	xxxxxxxx	= 6383 (Auth Blocked)

Table 6.19: Test Vector Fungsi Command Handler External Authentication

Name	Command_ExternalAuth
Input	-
Output	-

Table 6.20: Prototype Fungsi Command Handler External Authentication

```
12  Transmission_GetData(encrypted, CRYPT_BLOCK_LEN];
13
14  uint8_t retries;
15  switch( State_VerifyAuth( &encrypted ) )
16    {
17    case STATE_OK:
18      Response_SetSW( Response_OK , 0);
19      break;
20    case STATE_BLOCKED:
21      Response_SetSW( Response_CmdNotAllowed_AuthBlocked , 0);
22      break;
23    case STATE_WRONG:
24      retries = Memory_ReadByte(EXT_AUTH_RETRIES_ADDR);
25      Response_SetSW( Response_Warning_Counter | (retries & 0x0f) ,
```

```
            0);
26      }
27
28  }
```

Listing 6.10: Listing Program Fungsi Command Handler External Auth

6.10 Command Handler - Get Response

Berfungsi sebagai command handler untuk instruksi Get Response. Gambar 6.19 menampilkan DFD dari fungsi Command Handler Get Response. Diagram alir fungsi kemudian ditampilkan pada Gambar 6.20.

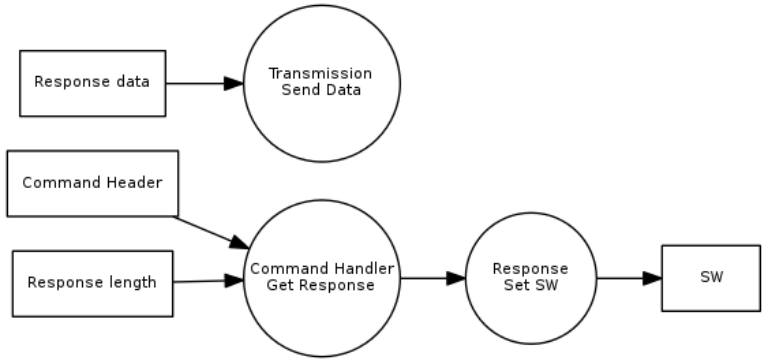

Figure 6.19: DFD Command Get Response

6.10.1 Pengujian

Input	**Output**	
Header	**Data**	**Status Word**
Setup : *response = [1,2,3,4,5]* *resplen = 5*		
[0x80,0xC0,0x00.0x00,0x05]	= [1,2,3,4,5]	= 6100 (Normal)
[0x80,0xC0,0x00.0x00,0x03]	= [1,2,3]	= 6100 (Normal)
[0x80,0xC0,0x00.0x00,0x08]	-	= 6700 (Wrong Length)

Table 6.21: Test Vector Fungsi Command Handler Get Response

Tabel 6.21 menampilkan Test Vector yang digunakan untuk menguji fungsi Command Handler Get Response.

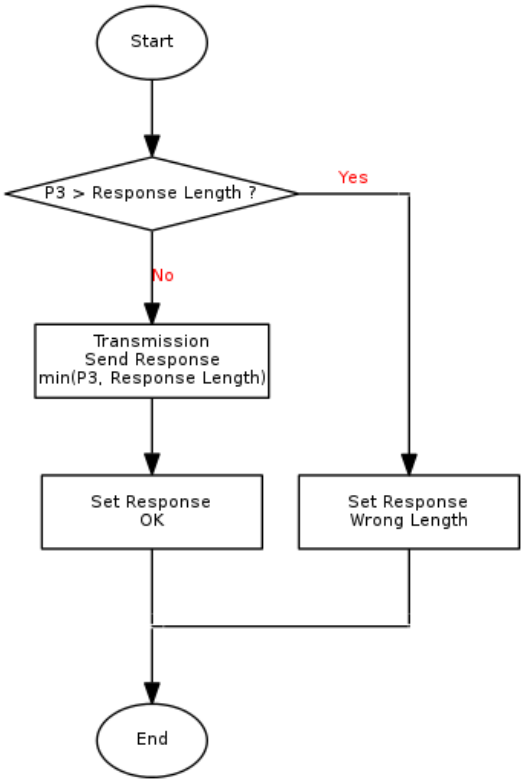

Figure 6.20: Flowchart Command Handler Get Response

6.10.2 Implementasi

Tabel 6.22 menampilkan purwarupa dari implementasi fungsi Command Handler Get Response.

Name	Command_GetResponse
Input	Command Header
Output	

Table 6.22: Prototype Fungsi Command Handler Get Response

Listing ?? menampilkan potongan program yang mengimplementasi fungsi Command Handler Get Response

```
1 void Command_GetResponse()
2 {
```

```
3   if( resplen==0 ) {
4     Response_SetSW( Response_CmdNotAllowed_ConditionNotSatisfied ,
          0);
5     return;
6   }
7
8   if( (header[4]>resplen) || (!header[4]) ) {
9     Response_SetSW( Response_WrongLength , 0);
10    return;
11  }
12  resplen=header[4];
13
14  /* ACK */
15  HAL_IO_TxByte( header[1] );
16
17  Transmission_SendData(response,resplen);
18
19  Response_SetSW( Response_Normal , 0);
20 }
```

Listing 6.11: Listing Program Fungsi Command Handler Get Response

Chapter 7

Response Manager

Response Manager utamanya berfungsi untuk menghasilkan Status Word (SW) yang sesuai dengan Response Type dari Command Handler, untuk kemudian dikirimkan oleh Transmission Handler SendSW. Table 7.1 menampilkan daftar Response Type dan Status Word yang sesuai.

Response Type	Status Word
OK	9000
Normal	6100
Counter	63C0
Wrong Length	6700
Not Supported	6800
Not Allowed	6900
Security Status	6982
Auth Blocked	6983
Condition Not Satisfied	6985
Wrong P1P2	6A00
File Not Found	6A82
INS Not Supported	6D00
CLA Not Supported	6E00
Fatal Error	6F00

Table 7.1: Daftar Response Type dan Status Word yang sesuai

Response Type sendiri dideklarasikan sebagai sebuah tipe data enumerasi sebagaimana ditunjukkan pada Listing 7.1, dan dapat ditemukan pada file header *response.h*.

```
1 typedef enum
2 {
```

```
  Response_OK,
  Response_Normal,
  Response_Counter,
  Response_WrongLength,
  Response_NotSupported,
  Response_CmdNotAllowed,
  Response_AuthBlocked,
  Response_ConditionNotSatisfied,
  Response_WrongP1P2,
  Response_FileNotFound,
  Response_INSNotSupported,
  Response_CLANotSupported,
  Response_FatalError,
} rspn_type;
```

Listing 7.1: Tipe Data Enumerasi Response Type

Beberapa Response Type dapat menerima parameter tambahan untuk menjelaskan response secara lebih terperinci. Sebagai contoh response type Normal dapat memuat parameter tambahan untuk menunjukkan jumlah byte yang masih tersedia sebagai response data untuk dapat diambil selanjutnya menggunakan command Get Response oleh Terminal. Response Type lainnya yang dapat menerima parameter tambahan ini adalah Counter, dengan parameter tambahan menunjukkan percobaan yang telah dilakukan dan gagal ketika melakukan verifikasi PIN. Status Word yang dihasilkan untuk response type dengan parameter tambahan ini adalah:

$$status\_word \mid extra\_parameter \tag{7.1}$$

Sebagai contoh untuk response type Counter ($63C0$) dengan parameter tambahan 2 (yang berarti telah gagal melakukan verifikasi sebanyak 2 kali), akan menghasilkan response type $63C2$.

Status Word ini kemudian akan disimpan dalam sebuah variabel unsigned integer 16 bit sebagai ditunjukkan pada Listing 7.2, dan dapat ditemukan pada file header *response.h*.

```
uint16_t sw;
```

Listing 7.2: Deklarasi variabel status word

7.1 Response Set SW

Berfungsi memetakan response type dengan Status Word (SW) yang sesuai. Gambar 7.1 menampilkan DFD dari fungsi Response Set SW. Diagram alir fungsi kemudian ditampilkan pada Gambar 7.2.

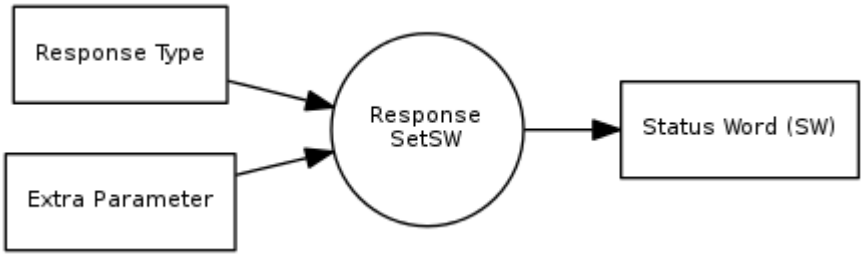

Figure 7.1: DFD Response Set SW

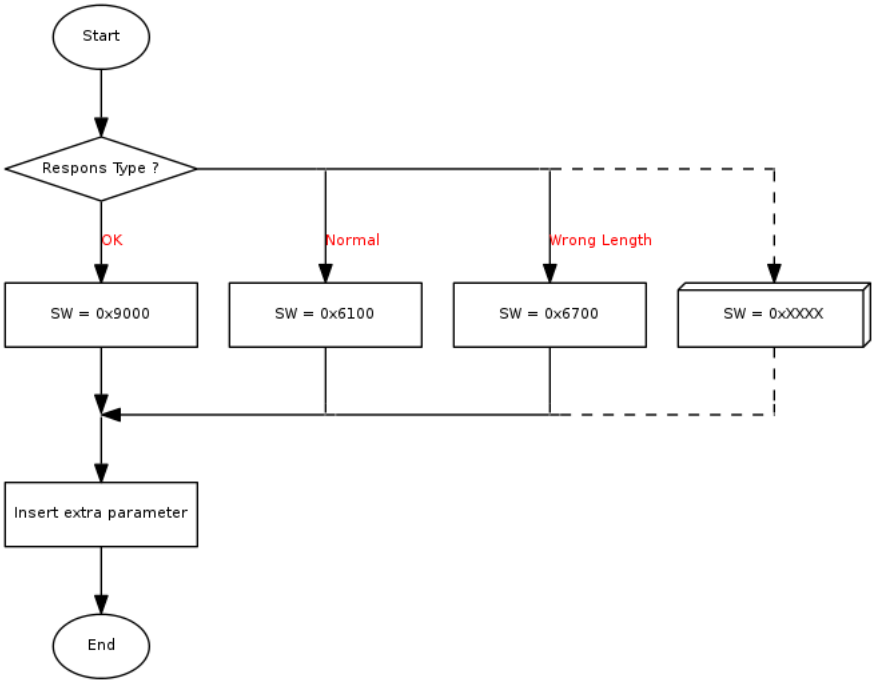

Figure 7.2: Flowchart Response Set SW

7.1.1 Pengujian

Tabel 7.2 menampilkan Test Vector yang digunakan untuk menguji fungsi Response Set SW.

Input		Output
response type	extra parameter	status word
OK	0	= 9000
Normal	0	= 6100
Counter	0	= 63C0
Wrong Length	0	= 6700
Not Supported	0	= 6800
Not Allowed	0	= 6900
Security Status	0	= 6982
Auth Blocked	0	= 6983
Condition Not Satisfied	0	= 6985
Wrong P1P2	0	= 6A00
File Not Found	0	= 6A82
INS Not Supported	0	= 6D00
CLA Not Supported	0	= 6E00
Fatal Error	0	= 6F00
Normal	0F	= 610F
Counter	03	= 63C3

Table 7.2: Test Vector Fungsi Response Set SW

7.1.2 Implementasi

Tabel 7.3 menampilkan purwarupa dari implementasi fungsi Set SW.

Name	Response_SetSW
Input	• response - Response Type (unsigned integer 8 bit) • extra - Extra Parameter (unsigned integer 8 bit)
Output	None

Table 7.3: Prototype Fungsi Response Set SW

Listing 7.3 menampilkan bagian program yang mengimplementasi fungsi Response Set SW, dan dapat ditemukan pada file *response.c*.

```
1 void Response_SetSW( rspn_type response, uint8_t extra )
2 {
3   switch( response )
4     {
```

```
5     case Response_OK:
6       sw = 0x9000;
7     case Response_Normal:
8       sw = 0x6100;
9     case Response_Warning_Counter:
10      sw = 0x63C0;
11    case Response_WrongLength:
12      sw = 0x6700;
13    case Response_NotSupported:
14      sw = 0x6800;
15    case Response_CmdNotAllowed:
16      sw = 0x6900;
17    case Response_CmdNotAllowed_SecurityStatus:
18      sw = 0x6982;
19    case Response_CmdNotAllowed_AuthBlocked:
20      sw = 0x6983;
21    case Response_CmdNotAllowed_ConditionNotSatisfied:
22      sw = 0x6985;
23    case Response_WrongP1P2:
24      sw = 0x6A00;
25    case Response_FileNotFound:
26      sw = 0x6A82;
27    case Response_INSNotSupported:
28      sw = 0x6D00;
29    case Response_CLANotSupported:
30      sw = 0x6E00;
31    case Response_FatalError:
32      sw = 0x6F00;
33    }
34
35  sw |= (uint16_t) extra;
36
37 }
```

Listing 7.3: Implementasi Fungsi Response Set SW

Chapter 8

File System

Bagian ini menjelaskan struktur data dan algoritma yang akan digunakan pada *file system*.

8.1 Data Structure

8.1.1 Konsep Block

Sebagaimana telah disebutkan pada File System, *file system* menggunakan layanan dari *Memory* untuk mengakses data file pada memory fisik. *Memory* menyediakan ruang penyimpanan virtual yang menggunakan pengalamatan linier dan kontinyu dimana setiap alamat dapat menyimpan data sebesar 1 byte. Namun penggunaan alamat untuk setiap 1 byte data ini kurang efisien dari sisi *file system* karena akan membutuhkan ruang yang besar untuk menyimpan alamat sebuah file. Sebagai contoh, untuk memory berukuran 128 kB sebagaimana banyak ditemukan pada *smart card* terbaru membutuhkan ruang untuk alamat sebanyak 3 byte (dengan asumsi unit penyimpanan terkecil 1 byte). Pada kenyataannya sebuah file umumnya juga memiliki ukuran yang jauh lebih besar. Maka, untuk meningkatkan efisiensi penggunaan ruang memory digunakan konsep *block* sebagai unit terkecil untuk mengalamati sebuah file dari sisi *file system*, dimana satu *block*x terdiri dari beberapa *byte* memory. Sebagai contoh, untuk ukuran *block* sebesar 4 byte, *block* `0` merujuk pada alamat `0x0000` hingga `0x0003`, *block* `1` merujuk pada alamat `0x0004` hingga `0x0007`, dan demikian seterusnya. Secara umum, alamat yang dirujuk dari sebuah *block* dapat diketahui menggunakan:

$$address = block\_number \cdot block\_size$$

Konsep *block* ini mengadopsi *file system* pada PC seperti Unix File System ataupun FAT. Namun berbeda dengan *file system* tersebut dimana block juga

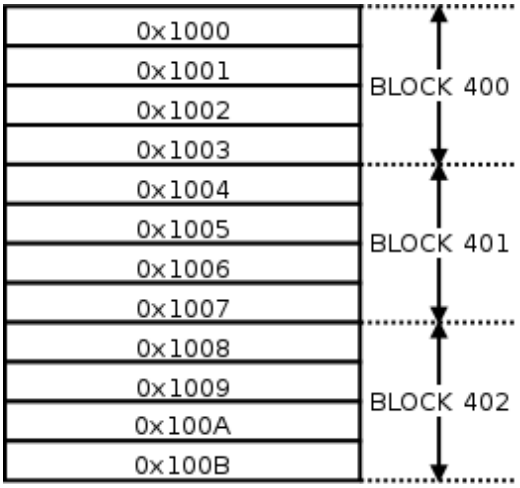

Figure 8.1: Konsep Block

menjadi unit operasi terkecil yang dapat dilakukan, pada *file system* yang dirancang, *block* hanya berfungsi sebagai unit pengalamatan terkecil. Operasi terhadap file tetap menggunakan unit *byte*. File system hanya akan membaca sejumlah *byte* yang diminta tidak perduli berapapun ukuran *block* yang digunakan. Demikian pula awal operasi pembacaan/penulisan dilakukan berdasarkan *offset* yang diukur dalam unit *byte*. Lebih jauh lagi penggunaan konsep *block* juga membantu dalam mengetahui bagian mana dari *virtual memory* yang telah digunakan (lebih lanjut akan dijelaskan pada **??**).

Disamping kelebihannya, penggunaan konsep *block* juga memiliki kelemahan yaitu pemborosan yang terjadi apabila ternyata hanya sebagian dari *block* yang digunakan untuk menyimpan data. Kerugian ini menjadi lebih terasa apabila menggunakan ukuran *block* yang besar. Sebagaimana telah disebutkan *block* merupakan unit terkecil pengalamatan file, maka ketika sebuah file membutuhkan ruang sebesar 9 byte, dengan ukuran *block* sebesar 8 byte, akan menggunakan sebanyak 2 *block* (= 16 *byte*). Selisih sebesar 7 (=16-9) *byte* ini kemudian menjadi percuma karena tidak dapat digunakan. Pada sistem komputer PC kerugian semacam ini memang tidak signifikan karena umumnya memiliki ruang penyimpanan yang relatif besar dan file yang besar pula, tetapi pada sistem *smart card* dengan penyimpanan terbatas hal ini tidak dapat diterima terutama ketika terjadi pada sejumlah besar file. Karenanya, penentuan ukuran *block* yang tepat menjadi penting untuk menghasilkan efisiensi yang paling tinggi.

8.1.2 Struktur file system

File-file pada *file system* akan saling terhubung satu sama lain untuk membentuk sebuah struktur *file system* (lihat Gambar 1.2). Hubungan ini dapat berupa *parent-to-child* dan sebaliknya *child-to-parent*. Hubungan *child-to-parent* merupakan hubungan satu-ke-satu sehingga tidak sulit untuk dideskripsikan dalam sebuah struktur data. Masalah kemudian muncul ketika mendeskripsikan hubungan *parent-to-child* yang merupakan hubungan satu-ke-banyak dan tidak ada batasan yang jelas mengenai jumlah anak dari sebuah file. Menyediakan sejumlah ruang yang telah ditentukan sebelumnya untuk menyimpan struktur ini jelas bukan pendekatan yang efektif karena akan membatasi jumlah anak yang mungkin dari sebuah file, serta tidak efisien mengingat ruang penyimpanan yang terbatas dari *smart card* dan karena mungkin saja hanya sebagaian kecil dari ruang tersebut yang digunakan. Menggunakan struktur yang dinamis juga bukan pendekatan yang baik pada sistem *smart card* yang terbatas karena dapat menyebabkan fragmentasi dan kerugian yang lebih besar.

Untuk mengatasi permasalahan tersebut maka diperkenalkan sebuah jenis hubungan baru yang dinamakan *sibling*, yaitu hubungan antara file *child* dengan *child* lainnya dalam sebuah directory (DF) yang sama. Dengan adanya hubungan ini maka sebuah DF (sebagai *parent*) hanya perlu menyimpan informasi satu file sebagai *child*, dan selanjutnya file *child* ini menunjuk satu file *child* lainnya sebagai *sibling*. Demikian seterusnya sehingga membentuk sebuah senarai berantai dari seluruh file dalam directory. Model pendekatan menggunakan *sibling* ini ditampilkan pada Gambar 8.2.

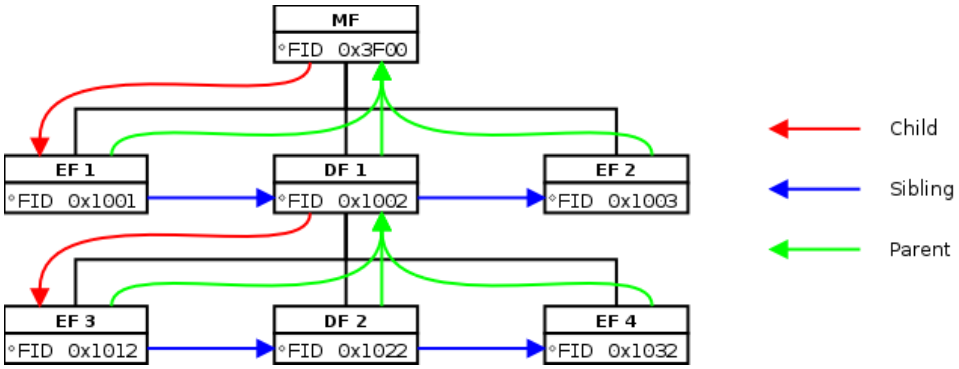

Figure 8.2: Pembagian file system

Sebagaimana dijelaskan pada Konsep Block, setiap unit data di dalam *file system* dialamati menggunakan nomor *block* yang berukuran 2 byte.

8.1.3 Bagian-bagian file system

Oleh *file system*, ruang memory virtual yang disediakan *Memory* dibagi menjadi tiga bagian besar, yaitu :

- Block Allocation Table (BAT)
- File Table
- File Body

Susunan ketiga bagian ini pada memory ditampilkan pada Gambar 8.3.

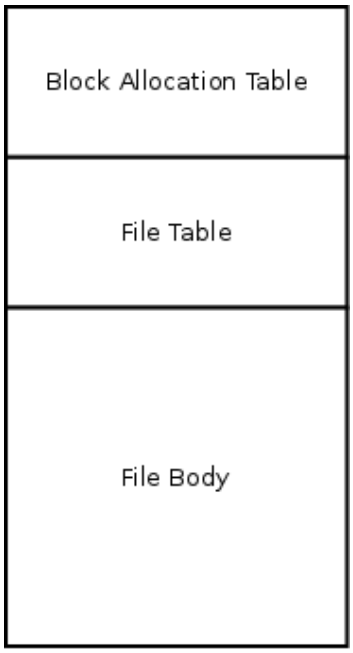

Figure 8.3: Pembagian file system

Block Allocation Table

Block Allocation Table (BAT) berfungsi menyimpan informasi mengenai *block* mana saja yang telah digunakan dan *block* mana yang belum. Setiap bit dari *block allocation table* merujuk pada sebuah *block*, dimana untuk *block* yang telah digunakan ditandai dengan nilai 1, sedangkan *block* yang tidak digunakan ditandai dengan nilai 0. Gambar 8.4 menampilkan contoh penggunaan *block allocation table* untuk menyimpan informasi penggunaan *block* pada *file*

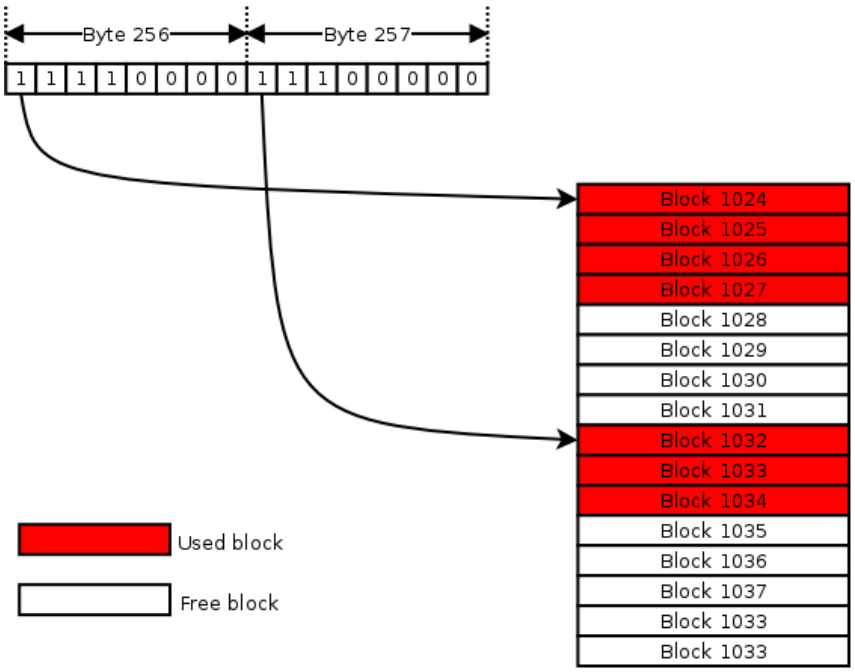

Figure 8.4: Contoh penggunaan block allocation table

system. *Block* yang dirujuk oleh sebuah bit pada *block allocation table* dapat dihitung menggunakan rumus :

$$block\_address = byte\_number \cdot 8 + bit\_number$$

Total ruang memory yang digunakan oleh *Block Allocation Table* dalam byte dapat dihitung mengggunakan rumus :

$$BAT\_size = (memory\_size/block\_size)/8$$

Sebagai contoh, untuk sistem dengan ruang memory penyimpanan sebesar 128 kB dan ukuran *block* 4 byte, maka besar ruang yang dibutuhkan untuk *block allocation table* adalah $128kB/4B = 32kbit = 4kB$.

Perlu diingat bahwa *block allocation table* akan mengurangi ruang memory virtual yang dapat digunakan untuk menyimpan informasi aktual. Tingkat kerugian penggunaan ruang memory yang diakibatkan oleh *block allocation memory* dapat dihitung sebagai perbandingan bagian memory yang digunakan untuk *block allocation table* terhadap seluruh ruang memory yang disediakan.

$$kerugian = BAT\_size/memory\_size = 8/block\_size$$

Terlihat bahwa kerugian akibat *block allocation table* berbanding terbalik dengan ukuran *block*. Karenanya, hal ini juga harus menjadi perhatian saat menentukan ukuran *block*.

File Table

File Table berfungsi menyimpan informasi mengenai struktur *file system* sebagai hubungan dari sejumlah file yang berawal dari sebuah MF (lihat Struktur file system). Informasi-informasi dasar dari setiap file disimpan dalam sebuah struktur data yang dinamakan *file header*. Sejumlah *file header* yang saling terhubung inilah yang kemudian disimpan di dalam file table untuk membentuk struktur *file system*.

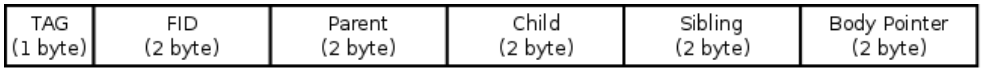

Figure 8.5: Struktur data file header

Gambar 8.5 menampilkan struktur akhir dari sebuah *file header*. Kemudian Gambar 8.6 menampilkan contoh struktur *file header* lengkap dari *file system* yang ditunjukkan pada Gambar 1.2. Susunan struktur *file header* ini pada *virtual memory* kemudian ditampilkan pada Gambar 8.7.

Tag Merupakan indikasi mengenai jenis sebuah file. (Lihat bagian Atribut File.)

File Type	Tag
MF	0x3F
DF	0x4F
EF	0x5F

Table 8.1: File Tag

FID Merupakan pengenal dari sebuah file dalam *smart card*. (Lihat bagian Atribut File.)

Parent Merupakan pointer yang menyimpan alamat dari file *parent*. (Lihat Struktur file system.)

Child Merupakan pointer yang menyimpan alamat dari file *child* pertama. (Lihat Struktur file system.)

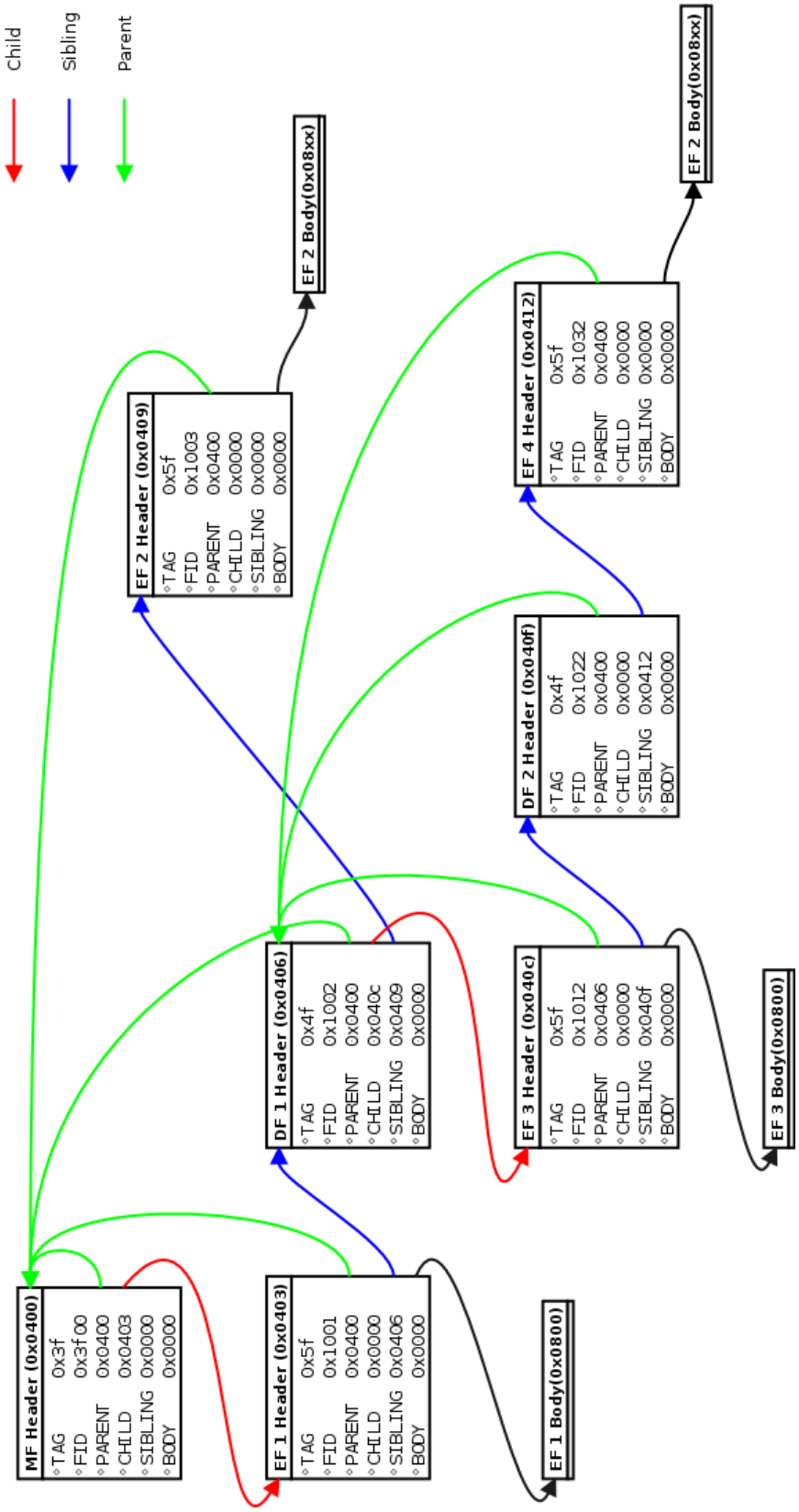

Figure 8.6: Struktur file header lengkap dari contoh sebuah file system (Gambar 1.2

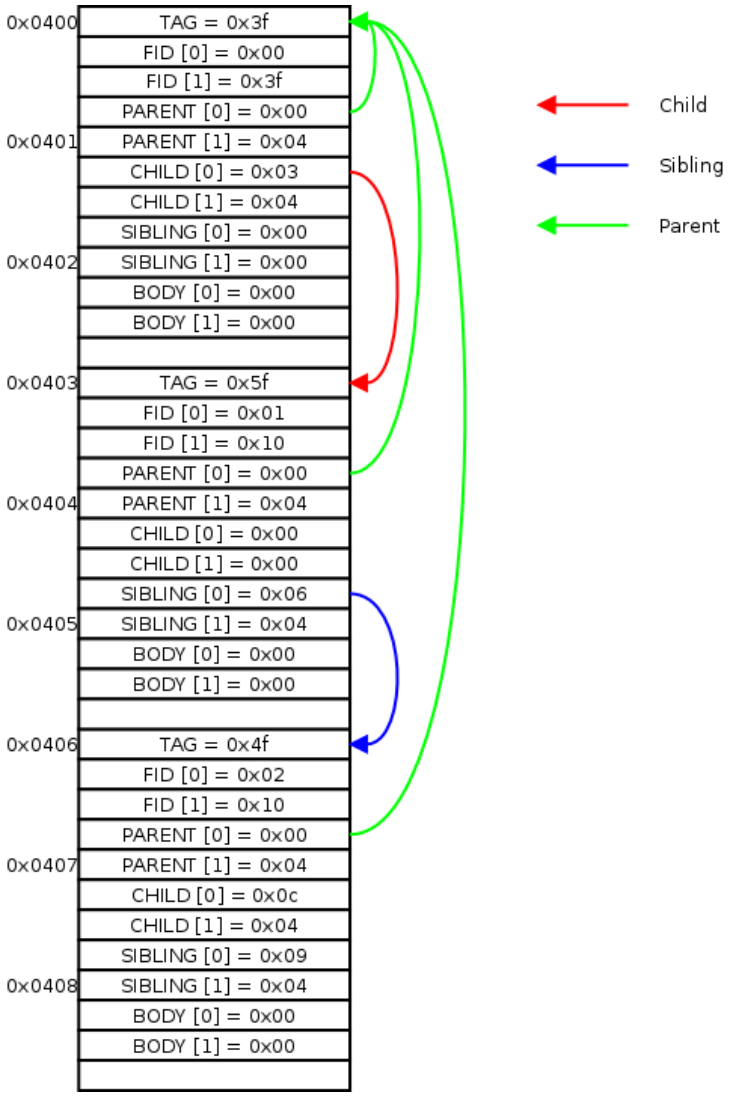

Figure 8.7: Susunan file header dari contoh sebuah file system (Gambar 1.2

Sibling Merupakan pointer yang menyimpan alamat dari file *sibling* berikutnya. (Lihat Struktur file system.)

File Body Merupakan pointer yang menyimpan alamat dari bagian file body dari file. (Mengenai file body ini akan dijelaskan lebih rinci pada bagian File Body.)

File Body

Bagian ini diperuntukkan bagi isi dari file yang sebenarnya. Isi file ini akan tergabung dalam sebuah struktur *file body* yang terdiri dari *File Body Header* dan *File Body Data*. *File Body Header* mengandung deskripsi dari isi *File Body Data* dan beberapa atribut lainnya yang terkait dengan File. Data aktual dari file sendiri disimpan pada *File Body Data*. Gambar 8.8 menampilkan struktur dari *File Body*

Figure 8.8: Struktur File Body

Gambar 8.9 menampilkan struktur data dari *File Body Header*.

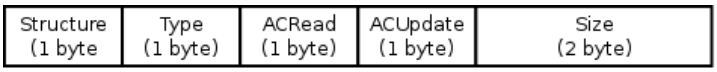

Figure 8.9: Pembagian file system

Structure Menunjukkan struktur file yang digunakan. Sesuai dengan standard ISO 7816-4, maka struktur file yang didukung terdiri dari : Transparent, Record, dan Cyclic. Setiap struktur file ini akan diberi tag sendiri untuk membedakannya dari struktur file lain sebagaimana ditampilkan pada Tabel **??**.

Type Menunjukkan tipe file. Sesuai dengan standard ISO 7816-4, sebuah file dapat berupa working file ataupun internal file. working file dapat dibaca oleh aplikasi dan sistem operasi, sedangkan internal file hanya dapat dibaca oleh sistem operasi. Sebagaimana File Structure, setiap File Type ini diberi Tag untuk membedakannya dari File Type lain yang ditunjukkan pada Tabel **??**.

Access cond. Menunjukkan tingkat keamanan yang harus dicapai sebelum melakukan operasi terhadap file, masing-masing AC read untuk operasi baca dan ACUpdate untuk operasi tulis. Nilai-nilai ini akan diverifikasi oleh State Manager.

Size Menunjukkan ukuran data file dalam byte.

8.2 Dokumentasi Fungsi

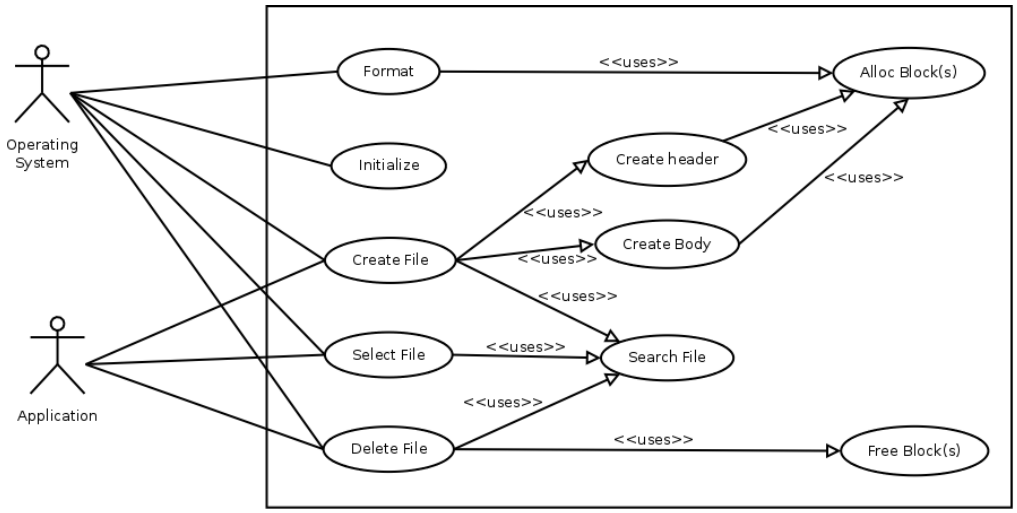

Figure 8.10: Use Case Diagram File System

Fungsi-fungsi yang terdapat pada *File System* dan hubungannya dapat dilihat melalui use case diagram pada Gambar 8.10. Dari usecase diagram tersebut terlihat fungsi-fungsi yang ada pada *file system* sebagai berikut :

Format Format the filesystem.

Initialize Initialize the filesystem.

Search File Search for a file

Select File Select a file

Create File Create a new file

Create Header Create a new file header

Create Body Create a new file body

Delete File Delete a file

Access Binary Access a transparent file

Access Record Operate on a record file

Alloc Block(s) Alloc block(s) of memory

Free Block(s) Free block(s) of memory

8.2.1 Format

Fungsi *Format* akan mempersiapkan memory untuk penggunaan pertama file system. Fungsi ini hanya sekali dipanggil oleh system operasi ketika finalisasi *smart card*.

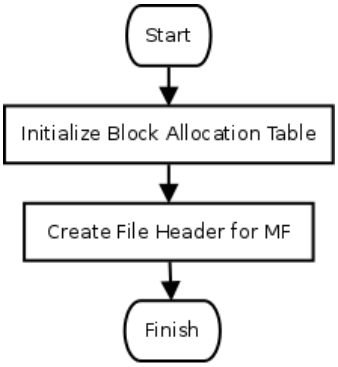

Figure 8.11: Algoritma fungsi format

Gambar 8.11 menampilkan prosedur yang dijalankan oleh fungsi *Format*. Fungsi *Format* akan mempersiapkan memory dengan menghapus seluruh isi memory dan membagi memory menjadi bagian-bagian sebagaimana dijelaskan pada Bagian Bagian-bagian file system. Setelah itu fungsi akan membuat file root (MF) di alamat pertama pada *File Table*.

Pengujian

Tabel 8.2 menampilkan Test Vector yang digunakan untuk menguji fungsi File System Format.

Implementasi

Tabel 8.3 menampilkan purwarupa dari implementasi fungsi File System Format. Fungsi ini akan diimplementasikan sebagai FSFormat() dan tidak

Output
Memory value
$FS\_ALLOC\_TABLE\_OFFSET \rightarrow FS\_ALLOC\_TABLE\_SIZE$ [0x00, ... 0x00]

Table 8.2: Test Vector Fungsi File System Format

menerima argumen apapun. Hasil keluaran dari fungsi adalah keterangan apakah fungsi berhasil atau mengalami kesalahan (error).

Name	FS_Format
Input	None
Output	Status

Table 8.3: Prototype Fungsi Format

Listing 8.1 menampilkan potongan program yang mengimplementasi fungsi FS Format.

```
1  int FS_Format()
2  {
3    uint16_t i;
4    uint16_t headerBlock;
5    uint16_t none16 = FS_NONE;
6    uint16_t fidMF = 0x3F00;
7    uint8_t tagMF  = FS_TAG_MF;
8    int status;
9
10   //Initialize Block Allocation Table
11   for(i=FS_ALLOC_TABLE_OFFSET;
         i<(FS_FILE_TABLE_OFFSET*FS_BLOCK_SIZE); i++)
12     Mem_WriteByte(FS_START + i, 0);
13
14   status = FS_ALLOC_HEADER(&headerBlock);
15
16   if(status == FS_OK && headerBlock == FS_FILE_TABLE_OFFSET){
17     FS_SET_HEADER_TAG(headerBlock, &tagMF);
18     FS_SET_HEADER_FID(headerBlock, &fidMF);
19     FS_SET_HEADER_PARENT(headerBlock, &headerBlock);
```

```
20      FS_SET_HEADER_CHILD(headerBlock, &none16);
21      FS_SET_HEADER_SIBLING(headerBlock, &none16);
22
23      return FS_OK;
24    }
25    else {
26      return FS_ERROR;
27    }
28  }
```

Listing 8.1: Listing Program Fungsi FS Format

8.2.2 Initialize

Fungsi *Initialize* akan mempersiapkan memory untuk dapat dipergunakan. Fungsi ini dipanggil oleh sistem operasi setiap kali *memory card* diberi daya dan memasuki tahap initialisasi.

Gambar 8.12 menampilkan prosedur yang dijalankan oleh fungsi *Initialize*.

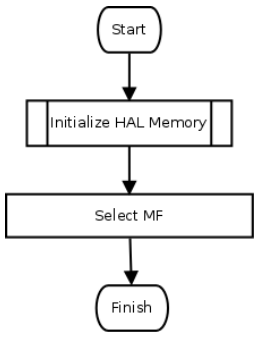

Figure 8.12: Algoritma fungsi Initialize

Pengujian

Output
State
CurrentFile = 0x0000

Table 8.4: Test Vector Fungsi File System Initialize

Tabel 8.4 menampilkan Test Vector yang digunakan untuk menguji fungsi File System Initialize.

Implementasi

Tabel 8.5 menampilkan purwarupa dari implementasi fungsi File System Initialize. Fungsi ini akan diimplementasikan sebagai FSInitialize() dan tidak menerima argumen apapun. Hasil keluaran dari fungsi adalah status apakah fungsi berhasil atau mengalami kesalahan (error).

Name	FS_Init
Input	None
Output	Status

Table 8.5: Prototype Fungsi FS Initialize

Listing 8.2 menampilkan potongan program yang mengimplementasi fungsi FS Initialize.

```
1 int FS_Init()
2 {
3   FSSelectMF();
4
5   return FS_OK;
6 }
```

Listing 8.2: Listing Program Fungsi FS Initialize

8.2.3 Create File

Fungsi *Create File* akan menciptakan sebuah file baru pada struktur *file system*. Fungsi ini dapat dipanggil oleh sistem operasi maupun aplikasi.

Gambar 8.13 menampilkan prosedur yang dijalankan oleh fungsi Create File. File baru hanya dapat dibuat apabila file yang sedang terpilih adalah sebuah DF, dimana file baru ini akan diletakkan dibawahnya (sebagai anak) pada struktur file system. Selain itu tidak boleh ada file lainnya dalam jangkauan parent yang memiliki FID yang sama dengan file baru agar tidak menimbulkan ambiguitas.

Apabila semua syarat telah terpenuhi, Sebuah file baru akan dibuat dengan menghasilkan sebuah file header dengan memanggil fungsi *Create Header*. Apabila fungsi *Create Header* berhasil membuat file header baru, maka prosedur dilanjutkan dengan memperbaharui file header dari *parent* atau *sibling*, tergantung apakah file baru tersebut merupakan *child* pertama atau tidak

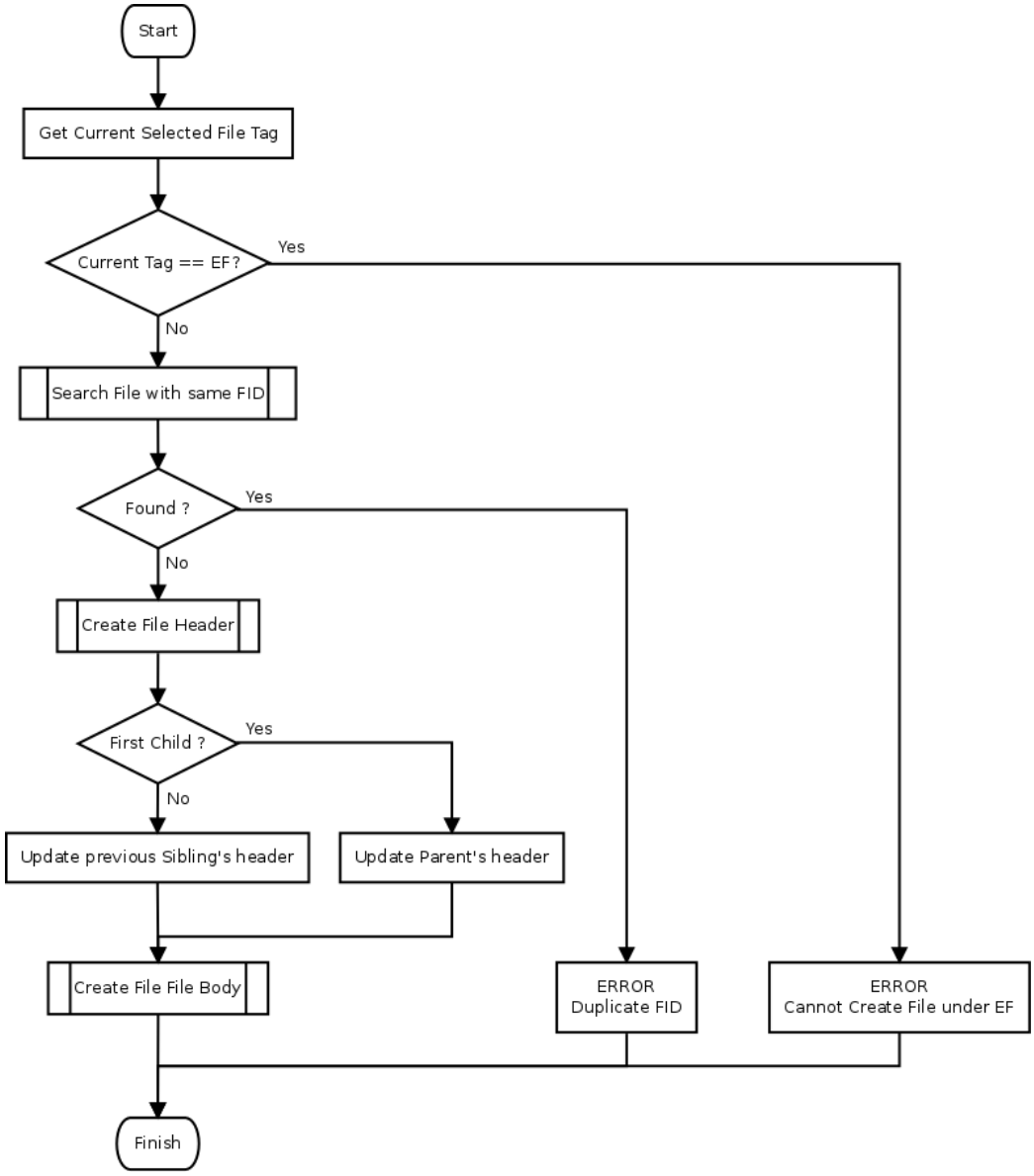

Figure 8.13: Algoritma fungsi Create File

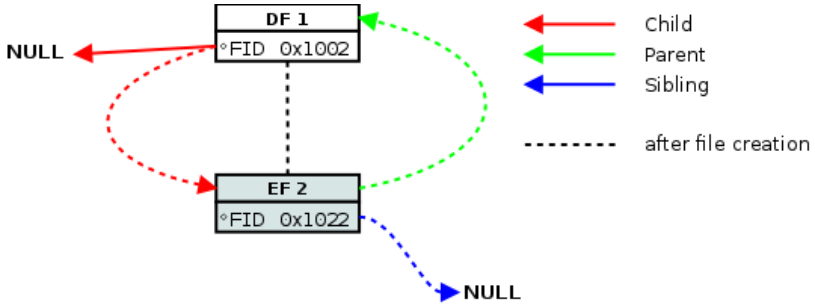

Figure 8.14: Contoh fungsi Create File (kasus *child* pertama)

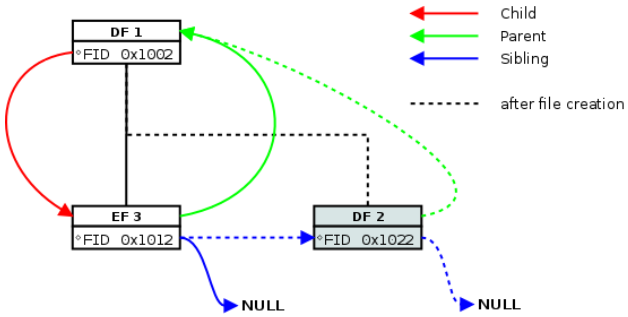

Figure 8.15: Contoh fungsi Create File (kasus *child* kedua)

dari *parent* file. Pada kasus *child* pertama, maka yang perlu diperbaharui adalah pointer *child* dari *parent* file (lihat Gambar 8.14). Sementara pada kasus *child* kedua dan seterusnya, maka yang diperbaharui adalah pointer *sibling* dari file *sibling* sebelumnya (lihat Gambar 8.15).

Setelah file header dari file parent/sibling diperbaharui, fungsi kemudian akan membuat sebuah file body (dengan memanggil fungsi Create Body) dengan atribut yang sesuai dengan file descriptor yang diberikan pada masukan fungsi. Apabila file body berhasil dibuat, maka file header yang telah dibuat sebelumnya akan diperbaharui dengan memasukkan alamat file body. Namun apabila file body tidak berhasil dibuat, maka file header yang telah dibuat sebelumnya harus dihapus dengan memanggil fungsi Delete File.

Pengujian

Input		Output	
Tag	**File descriptor**	**Memory value**	**Return Value**
0x5f	$EF\_st \rightarrow$ FID = 0x1234 structure = TRANSPARENT type = WORKING ACRead = 0x00 ACUpdate = 0x01 size = 0xff	$HEADER\_TAG \rightarrow 0x5f$ $HEADER\_FID \rightarrow 0x1234$ $BODY\_STRUCTURE \rightarrow 0$ $BODY\_TYPE \rightarrow 0$ $BODY\_ACREAD \rightarrow 0x00$ $BODY\_ACUPDATE \rightarrow 0x01$ $BODY\_SIZE \rightarrow 0xff$	FS_OK

Table 8.6: Test Vector Fungsi File System Create File

Tabel 8.6 menampilkan Test Vector yang digunakan untuk menguji fungsi File System Create File.

Implementasi

Tabel 8.7 menampilkan purwarupa dari implementasi fungsi File System Create File. Fungsi ini akan diimplementasikan sebagai FSCreateFile().

Name	FS_CreateFile
Input	File descriptor
Output	Status

Table 8.7: Prototype Fungsi FS Create File

Listing 8.3 menampilkan potongan program yang mengimplementasi fungsi FS Create File.

```
1  int FS_CreateFile(int tag, void * desc)
2  {
3    uint16_t header, body;
4    uint16_t current;
5    uint16_t fid;
6    uint8_t currentTag;
7    int status;
8
9    if(tag == FS_TAG_DF){
10     fid = ((struct DF_st *)desc)->FID;
11   }
12   else if (tag == FS_TAG_EF) {
13     fid = ((struct EF_st *)desc)->FID;
14   }
15
16   //check if current is DF
17   current = State_GetCurrent();
18
19   FS_GET_HEADER_TAG(current, (uint8_t*)&currentTag);
20
21   if(currentTag == FS_TAG_EF){
22     return FS_ERROR;
23   }
24
```

```
25   //check consistency
26
27   //check all FID of current DF
28   status = FSSearchFID(fid);
29
30   if(status != FS_NONE){
31     return FS_ERROR_DUPLICATE_FID;
32   }
33
34   //create file header
35   status = FSCreateHeader((uint8_t)tag, fid, &header);
36
37   if (status != FS_OK){
38     return status;
39   }
40
41   //create file body
42   if(tag == FS_TAG_EF) {
43     status = FSCreateBodyEF((struct EF_st *)desc, &body);
44
45     if (status != FS_OK){
46       return status;
47     }
48
49     FS_SET_HEADER_BODY(header, &body);
50
51   }
52
53   return 0;
54 }
```

Listing 8.3: Listing Program Fungsi FS Create File

8.2.4 Delete File

Fungsi *Delete File* akan menghapus sebuah file dari struktur file system. Fungsi ini dapat dipanggil oleh sistem operasi maupun aplikasi.

Gambar 8.16 menampilkan prosedur yang dijalankan oleh fungsi Delete File. Pertama, file yang akan dihapus dicari dengan memanggil fungsi Search File. Setelah ditemukan, file header dari parent / sibling diperbarui sehingga tidak lagi menunjuk pada file yang akan dihapus. Gambar 8.17-8.19 menampilkan beberapa contoh kasus penghapusan file.

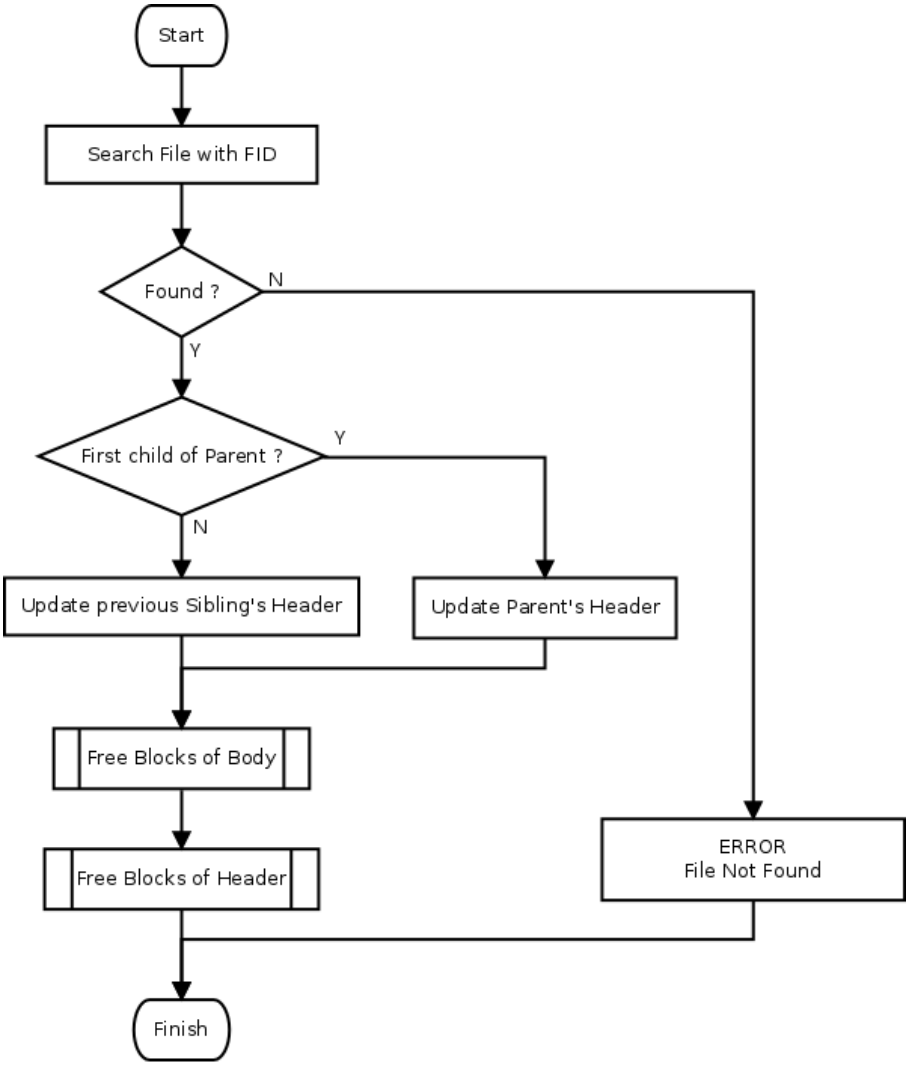

Figure 8.16: Algoritma fungsi Delete File

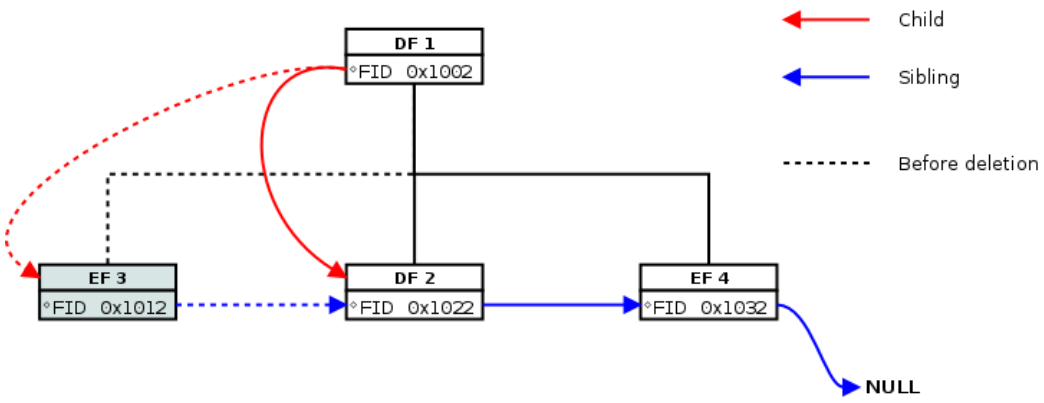

Figure 8.17: Contoh penghapusan file (kasus anak pertama)

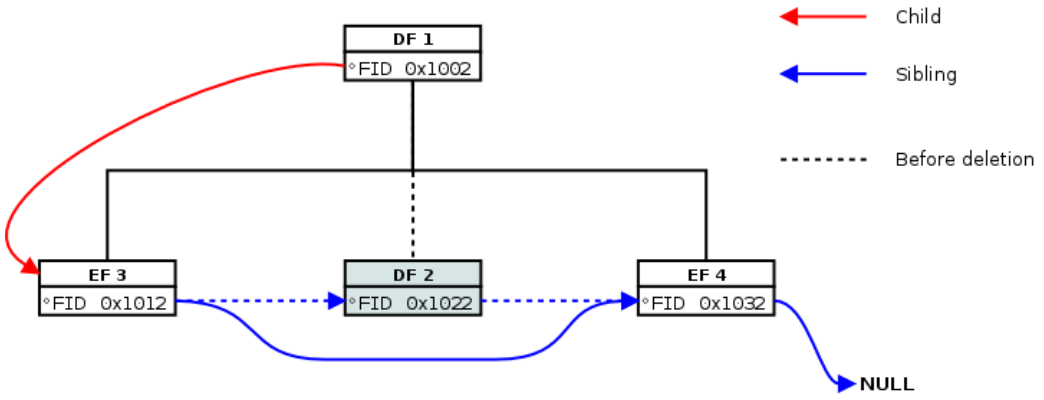

Figure 8.18: Contoh penghapusan file (kasus bukan anak pertama/terakhir)

Pada kasus pertama (Gambar 8.17), file yang dihapus (EF3) merupakan anak pertama dari file parent (DF1). Untuk itu perlu dilakukan pembaruan pada pointer child dari file parent header sehingga menunjuk pada file sibling dari file yang akan dihapus (DF2).

Pada kasus kedua (Gambar 8.18), file yang dihapus (DF2) bukan merupakan anak pertama dan bukan pula yang terakhir dari file parent (DF1). Untuk itu perlu dilakukan pembaruan pada pointer sibling dari file sibling sebelumnya (EF3) sehingga menunjuk pada file sibling dari file yang akan dihapus (EF4).

Pada kasus ketiga (Gambar 8.19), file yang dihapus (EF4) merupakan anak yang terakhir dari file parent (DF1). Untuk itu perlu dilakukan pembaruan pada pointer sibling dari file sibling sebelumnya (DF2). Karena EF4 tidak memiliki sibling berikutnya, maka pointer sibling dari file sibling DF2 akan menunjuk ke NULL.

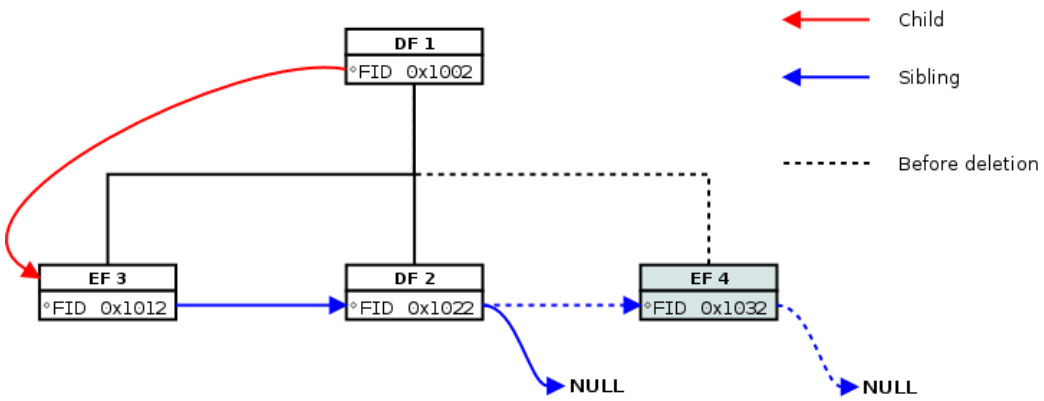

Figure 8.19: Contoh penghapusan file (kasus anak terakhir)

Pengujian

Input	Output	
FID	**Memory value**	**Return Value**
0x1234	$HEADER\_CHILD \rightarrow 0x0000$	FS_OK

Table 8.8: Test Vector Fungsi File System Delete File

Tabel 8.8 menampilkan Test Vector yang digunakan untuk menguji fungsi File System Delete File.

Implementasi

Tabel 8.9 menampilkan purwarupa dari implementasi fungsi File System Delete File. Fungsi ini akan diimplementasikan sebagai FSDeleteFile().

Name	FSDeleteFile
Input	FID
Output	Status

Table 8.9: Prototype Fungsi Delete File

Listing 8.4 menampilkan potongan program yang mengimplementasi fungsi FS Delete File.

```
int FSDeleteFile(uint16_t fid)
{
  uint16_t header, parent, sibling, child, siblingBefore,
      siblingNext;
  uint16_t body, bodySize;
  uint8_t tag;

  //check FID
  header = FSSearchFID(fid);

  if(header == FS_NONE){
    return FS_ERROR_NOT_FOUND;
  }

  //if DF, make sure it doesn't have any child

  //if first child, untie from parent, and change parent's child to
      sibling (if any)
  FS_GET_HEADER_PARENT(header, &parent);

  FS_GET_HEADER_CHILD(parent, &sibling);

  if(sibling == header) {
    FS_GET_HEADER_SIBLING(header, &sibling);
    FS_SET_HEADER_CHILD(parent, &sibling);
  }
  //else, change the sibling before to chain to sibling after
  else {

    while(sibling != header) {
      siblingBefore = sibling;
      FS_GET_HEADER_SIBLING(sibling, &sibling);
    }

    FS_GET_HEADER_SIBLING(header, &siblingNext);
    FS_SET_HEADER_SIBLING(siblingBefore, &siblingNext);
  }

  //free body
  FS_GET_HEADER_TAG(header, &tag);

  if(tag == FS_TAG_EF){
    FS_GET_HEADER_BODY(header, &body);
```

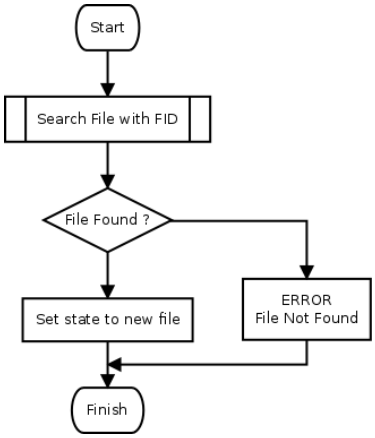

Figure 8.20: Algoritma fungsi Select File

```
42      FS_GET_BODY_SIZE(body, &bodySize);
43
44      FSFree(body,
            CEIL(((FS_BODY_HEADER_SIZE)+bodySize),FS_BLOCK_SIZE));
45    }
46
47    //free header
48    FSFree(header, CEIL((FS_HEADER_SIZE),FS_BLOCK_SIZE));
49
50    return FS_OK;
51  }
```

Listing 8.4: Listing Program Fungsi FS Create File

8.2.5 Select File

Fungsi Select File akan mengganti state file yang sedang terpilih dari smart card menjadi file dengan FID yang diberikan.

Gambar 8.20 menampilkan prosedur yang dijalankan oleh fungsi Select File. Pertama, file yang akan dipilih dicari dengan memanggil fungsi Search File. Apabila file ditemukan, maka state saat ini akan diubah sehingga menunjuk pada file yang dipilih. Sebaliknya apabila tidak ditemukan maka akan mengembalikan nilai kesalahan *File Not Found*.

Input	Output	
FID	**State**	**Return Value**
0x1234	$CurrentFile \rightarrow HEADER$	FS_OK

Table 8.10: Test Vector Fungsi File System Select File

Pengujian

Tabel 8.10 menampilkan Test Vector yang digunakan untuk menguji fungsi File System Select File.

Implementasi

Tabel 8.11 menampilkan purwarupa dari implementasi fungsi File System Select File. Fungsi ini akan diimplementasi oleh FSSelectFile().

Name	FS_SelectFID
Input	FID
Output	Status

Table 8.11: Prototype Fungsi Select File

Listing 8.5 menampilkan potongan program yang mengimplementasi fungsi FS Select File.

```
uint16_t FS_SelectFID(uint16_t fid)
{
  uint16_t file;

  file = FSSearchFID(fid);

  if( file == FS_NONE )
    return FS_ERROR_NOT_FOUND;

  State_SetCurrent(file);

  return FS_OK;
}
```

Listing 8.5: Listing Program Fungsi FS Select File

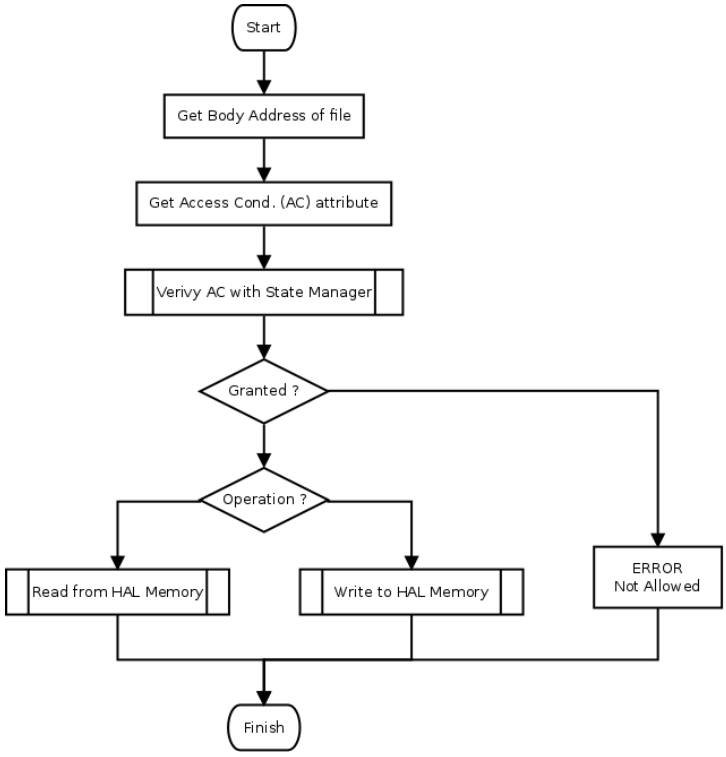

Figure 8.21: Algoritma fungsi Access Binary

8.2.6 Access Binary

Fungsi *Access Binary* dipanggil oleh sistem operasi atau aplikasi untuk mengakses file EF bertipe transparent. Sebelum memanggil fungsi Access Binary maka sistem operasi atau aplikasi harus memilih file yang akan diakses menggunakan fungsi *File Select*.

Terdapat dua jenis operasi yang dapat dilakukan, yaitu Read dan Update. Operasi Read akan membaca data dari file yang sedang dipilih dan mengembalikan ke fungsi pemanggil. Sementara Operasi Update akan mengubah data pada file dengan data yang berasal dari fungsi pemanggil. Fungsi ini akan diimplementasikan oleh FSAccessBinary().

Gambar 8.21 menampilkan prosedur yang dilakukan fungsi access binary. Pertama, fungsi akan mengambil alamat body dari file yang disimpan pada file header. Lalu dari file body tersebut fungsi akan mengambil *access condition* yang diperlukan untuk operasi dan diverifikasi ke *state manager*. Apabila *state manager* menyetujui akses, fungsi akan dilanjutkan dengan melakukan operasi yang diminta (baca/tulis). Sebaliknya apabila *state manager* tidak menyetujui, fungsi akan mengembalikan status kesalahan *Operation Not Al-*

lowed.

Pengujian

Input				Output	
Operation	**Offset**	**Length**	**Data**	**Memory**	**D**
UPDATE	0x0000	0x00ff	$[0x00 \rightarrow 0xff]$	$BODY\_BODY == [0x00 \rightarrow 0xff]$	
READ	0x0000	0x00ff		$BODY\_BODY == [0x00 \rightarrow 0xff]$	$[0x00$

Table 8.12: Test Vector Fungsi File System Access Binary

Tabel 8.12 menampilkan Test Vector yang digunakan untuk menguji fungsi File System Access Binary.

Implementasi

Tabel 8.13 menampilkan purwarupa dari implementasi fungsi File System Access Binary.

Name	FSAccessBinary
Input	• Operation (Read or Update) • Offset (in bytes) • Lenght of data to read/update • Data to update
Output	• Data readed • Status

Table 8.13: Prototype Fungsi Access Binary

Listing 8.6 menampilkan potongan program yang mengimplementasi fungsi FS Access Binary.

```
1 int FSAccessBinary(int op, int offset, int length, uint8_t
     *databyte)
2 {
3   uint16_t header, body;
```

```
4
5   header = State_GetCurrent();
6
7   FS_GET_HEADER_BODY(header,&body);
8
9   if(op == FS_OP_READ){
10    if(length == 0)
11      FS_GET_BODY_SIZE(body, &length);
12
13    if(length > 256)
14      length = 256;
15
16    FS_GET_BODY_BODY(body, length, databyte);
17  }
18  else if(op = FS_OP_UPDATE){
19    FS_SET_BODY_BODY(body, length, databyte);
20  }
21
22  return length;
23 }
```

Listing 8.6: Listing Program Fungsi FS Access Binary

8.2.7 Access Record

Fungsi *Access Record* dipanggil oleh sistem operasi atau aplikasi untuk mengakses file EF bertipe Record. Sebelum memanggil fungsi *Access Record* maka sistem operasi atau aplikasi harus memilih file yang akan diakses menggunakan fungsi *File Select*.

Terdapat lima jenis operasi yang dapat dilakukan, yaitu Read, Update, Append Record, Insert Record, dan Delete Record. Operasi Read akan membaca data record dari file yang sedang dipilih dan mengembalikan ke fungsi pemanggil. Operasi Update akan mengubah data record pada file dengan data record yang berasal dari fungsi pemanggil. Operasi Append akan menambahkan sebuah record baru setelah record yang paling akhir. Operasi Insert akan menambahkan sebuah record baru diantara record yang ada. Operasi Delete akan menghapus sebuah record dari file. Fungsi ini akan diimplementasikan oleh FSAccessRecord().

Input				Output	
Operation	**Offset**	**Length**	**Data**	**Memory**	**Data**

Table 8.14: Test Vector Fungsi File System Access Record

Pengujian

Tabel 8.14 menampilkan Test Vector yang digunakan untuk menguji fungsi File System Access Record.

Implementasi

Tabel 8.15 menampilkan purwarupa dari implementasi fungsi File System Access Record.

Name	FSAccessRecord
Input	• Operation (Read/Update/Append/Insert/Delete) • Record No. (for all operation except Append Record) • Lenght of record data (for all operation except Delete Record) • Record data (for all operation except Delete Record)
Output	• Data readed (for Read Operation) • Status

Table 8.15: Prototype Fungsi Access Record

8.2.8 Search File

Fungsi *Search File* dipanggil oleh fungsi-fungsi File System lainnya untuk menemukan file dengan FID yang diberikan. Fungsi ini akan diimplementasikan sebagai FSSearchFile, dan akan mengembalikan alamat block dari

file header dari file yang dicari apabila ditemukan, atau NULL apabila file tidak ditemukan.

Gambar 8.22 menampilkan prosedur yang dijalankan oleh fungsi Search File. Pencarian akan dilakukan dimulai dari parent file dari file yang sedang terpilih, dilanjutkan dengan memeriksa setiap sibling, dan berikutnya memeriksa seluruh child apabila file yang terpilih adalah sebuah DF.

Pengujian

Input	**Output**	
FID	**Address**	**Return Value**
0x1234	BLOCK ADDRESS	FS_OK

Table 8.16: Test Vector Fungsi File System Search File

Tabel 8.16 menampilkan Test Vector yang digunakan untuk menguji fungsi File System Search File.

Implementasi

Tabel 8.17 menampilkan purwarupa dari implementasi fungsi File System Search File.

Name	FSSearchFile
Input	FID
Output	• Block address of file header • Status (if Error)

Table 8.17: Prototype Fungsi Search File

Listing 8.7 menampilkan potongan program yang mengimplementasi fungsi FS Search File.

```
1 uint16_t FSSearchFID(uint16_t fid)
2 {
3   uint16_t current, parent, sibling, child;
4   uint16_t tempFID;
5   uint8_t tempTag;
```

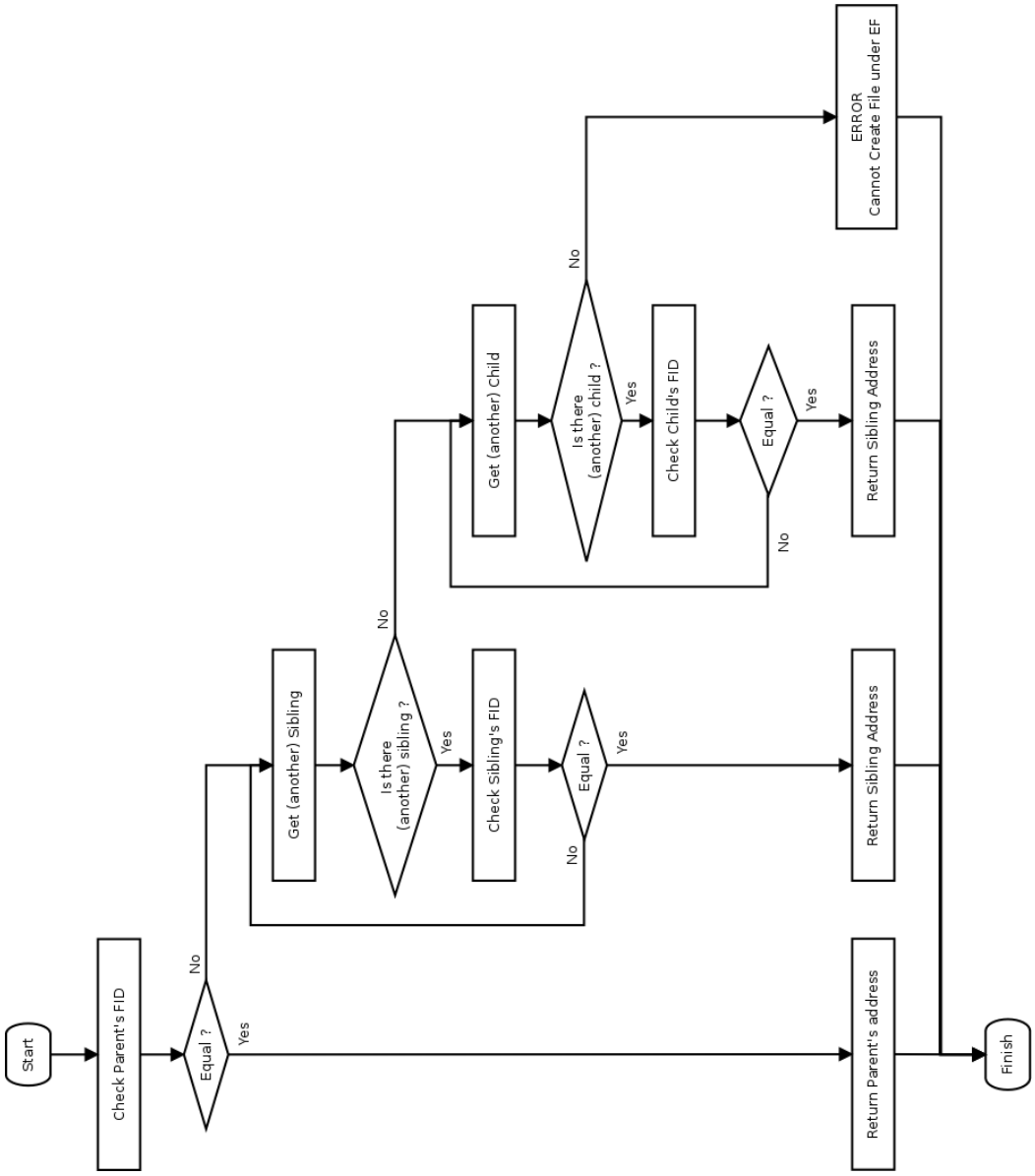

Figure 8.22: Algoritma fungsi Search File

```
6
7   current = State_GetCurrent();
8
9   //check parent
10  FS_GET_HEADER_PARENT(current, &parent);
11
12  FS_GET_HEADER_FID(parent, &tempFID);
13
14  if(tempFID == fid){
15    return parent;
16  }
17
18  //check sibling
19  FS_GET_HEADER_CHILD(parent, &sibling);
20
21  while(sibling != FS_NONE) {
22
23    FS_GET_HEADER_FID(sibling, &tempFID);
24
25    if(tempFID == fid){
26      return sibling;
27    }
28
29    FS_GET_HEADER_SIBLING(sibling, &sibling);
30
31  }
32
33  //check child
34
35  FS_GET_HEADER_TAG(current, &tempTag);
36
37  if(tempTag == FS_TAG_DF || tempTag == FS_TAG_MF) {
38
39    FS_GET_HEADER_CHILD(current, &child);
40
41    while (child != FS_NONE)
42    {
43
44      FS_GET_HEADER_FID(child, &tempFID);
45
46      if(tempFID == fid){
47        return child;
48      }
```

```
49
50      FS_GET_HEADER_SIBLING(child, &child);
51
52    }
53  }
54
55  return FS_NONE;
56 }
```

Listing 8.7: Listing Program Fungsi FS Search File

8.2.9 Create Header

Fungsi *Create Header* dipanggil oleh fungsi Create File untuk menghasilkan sebuah file header baru pada file table berdasarkan file descriptor yang diberikan.

Gambar 8.23 menampilkan prosedur yang dijalankan oleh fungsi Create Header. Pertama, fungsi akan mengalokasikan sejumlah block untuk header dengan memanggil fungsi Alloc Block(s). Apabila berhasil, dan sejumlah block telah dialokasikan, maka block tersebut akan diisi dengan struktur data file header sebagaimana yang telah dijelaskan pada bagian File Table. Tag dan FID yang digunakan diambil dari file descriptor, sementara parent merupakan file yang sedang dipilih sekarang yang diperoleh melalui State Manager. Child dan Sibling diisi nilai NULL karena file merupakan sebuah file baru sehingga tidak memiliki child dan sibling. Body pointer dibiarkan barena akan diisi kemudian setelah file body dihasilkan.

Pengujian

Input		**Output**	
TAG	**FID**	**Memory value**	**Return Value**
0x5f	0x1234	$HEADER\_TAG \rightarrow 0x5f$ $HEADER\_FID \rightarrow 0x1234$ $HEADER\_PARENT \rightarrow PARENT$ $HEADER\_CHILD \rightarrow 0x00$ $HEADER\_SIBLING \rightarrow 0x00$	FS_OK

Table 8.18: Test Vector Fungsi File System Create Header

Tabel 8.18 menampilkan Test Vector yang digunakan untuk menguji fungsi File System Create Header.

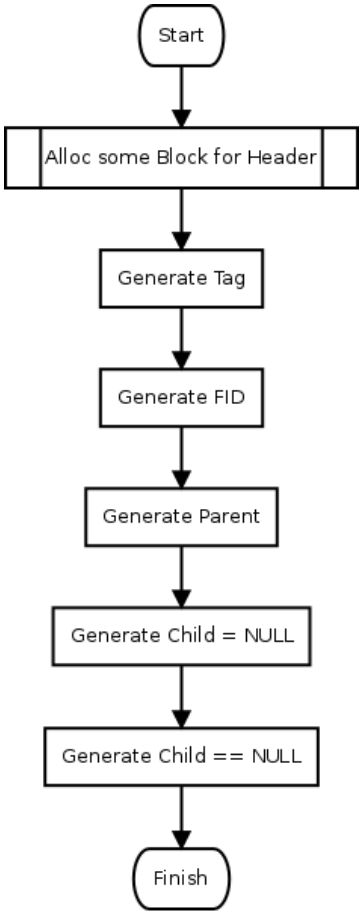

Figure 8.23: Algoritma fungsi Create Header

Implementasi

Tabel 8.19 menampilkan purwarupa dari implementasi fungsi File System Create Header. Fungsi ini akan diimplementasikan sebagai FSCreateHeader().

Name	FSCreateHeader
Input	• Tag • FID
Output	• Block address of header created • Status

Table 8.19: Prototype Fungsi Create Header

Listing 8.1 menampilkan potongan program yang mengimplementasi fungsi FS Format.

Listing 8.8 menampilkan potongan program yang mengimplementasi fungsi FS Create Header.

```
1  int FSCreateHeader(uint8_t tag, uint16_t fid, uint16_t * addr)
2  {
3    uint16_t     current, currentChild, currentChildSibling,
         currentChildTemp;
4    uint16_t     headerBlock;
5    uint16_t     none16 = (uint16_t)FS_NONE;
6    int          status;
7
8    current = State_GetCurrent();
9
10   //alloc some space for header
11   status = FS_ALLOC_HEADER(&headerBlock);
12
13   if(status == FS_ERROR_INSUFFICIENT_SPACE){
14     return status;
15   }
16
17   //update header
18   FS_SET_HEADER_TAG(headerBlock, &tag);
```

```
19   FS_SET_HEADER_FID(headerBlock, &fid);
20   FS_SET_HEADER_PARENT(headerBlock, &current);
21   FS_SET_HEADER_CHILD(headerBlock, &none16);
22   FS_SET_HEADER_SIBLING(headerBlock, &none16);
23
24   //update parent/sibling DF header
25   FS_GET_HEADER_CHILD(current, &currentChild);
26
27   if (currentChild == none16) {
28     FS_SET_HEADER_CHILD(current, &headerBlock);
29   }
30   else {
31
32     while (currentChild != FS_NONE){
33
34       currentChildTemp = currentChild;
35
36       FS_GET_HEADER_SIBLING(currentChild, &currentChild);
37
38     }
39
40     currentChild = currentChildTemp;
41
42     FS_SET_HEADER_SIBLING(currentChild, &headerBlock);
43
44   }
45
46   *addr = headerBlock;
47
48   return FS_OK;
49 }
```

Listing 8.8: Listing Program Fungsi FS Create Header

8.2.10 Alloc Block(s)

Fungsi *Alloc Block(s)* dipanggil oleh fungsi-fungsi lain untuk mengalokasikan sejumlah block untuk digunakan.

Gambar 8.24 menampilkan algoritma yang digunakan oleh fungsi Alloc Block(s). Untuk memudahkan memahami algoritma yang digunakan, pada Gambar 8.26 menampilkan contoh penggunaan fungsi Alloc Block(s) dengan parameter :

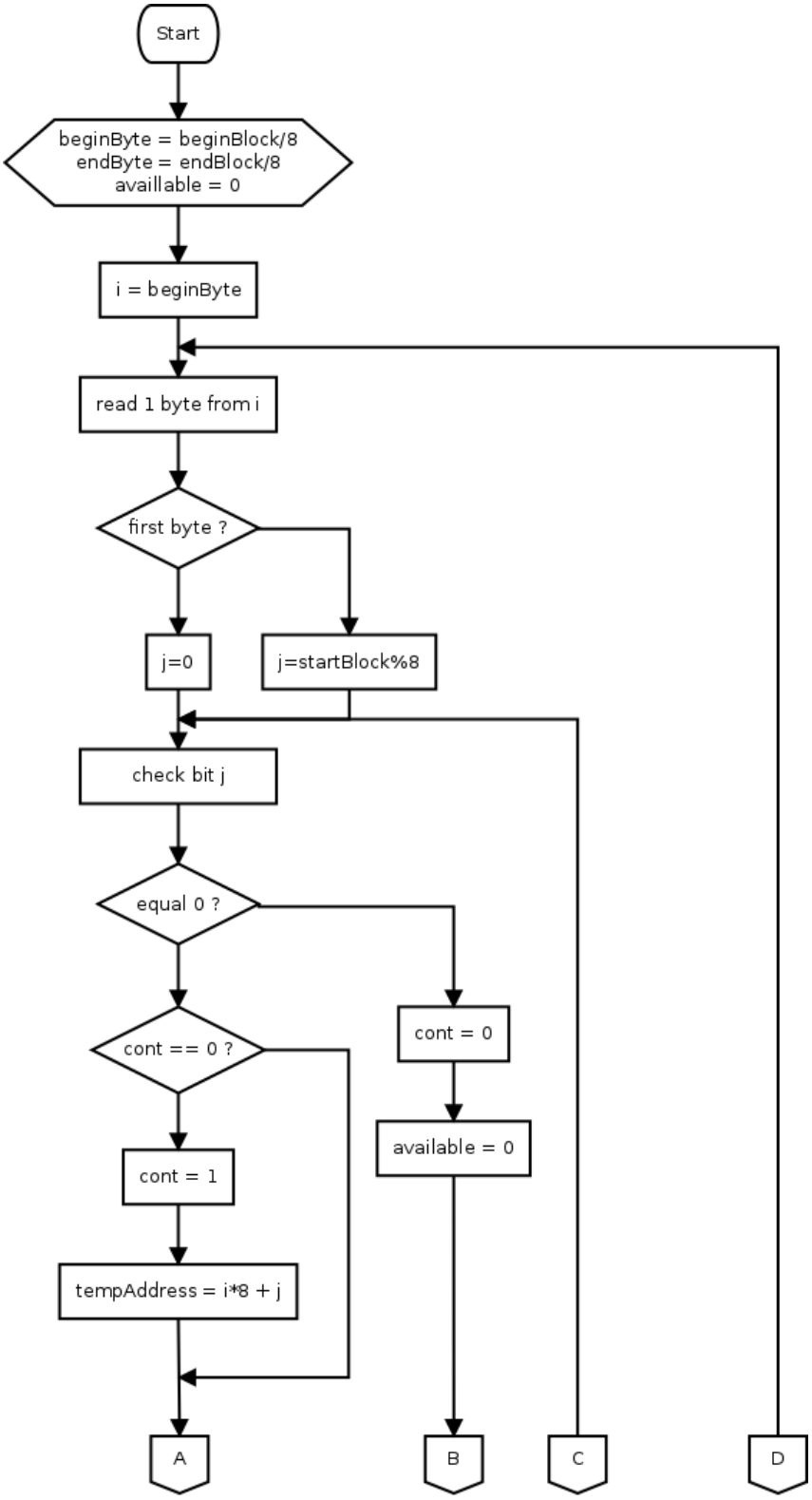

Figure 8.24: Algoritma fungsi Alloc Block(s)

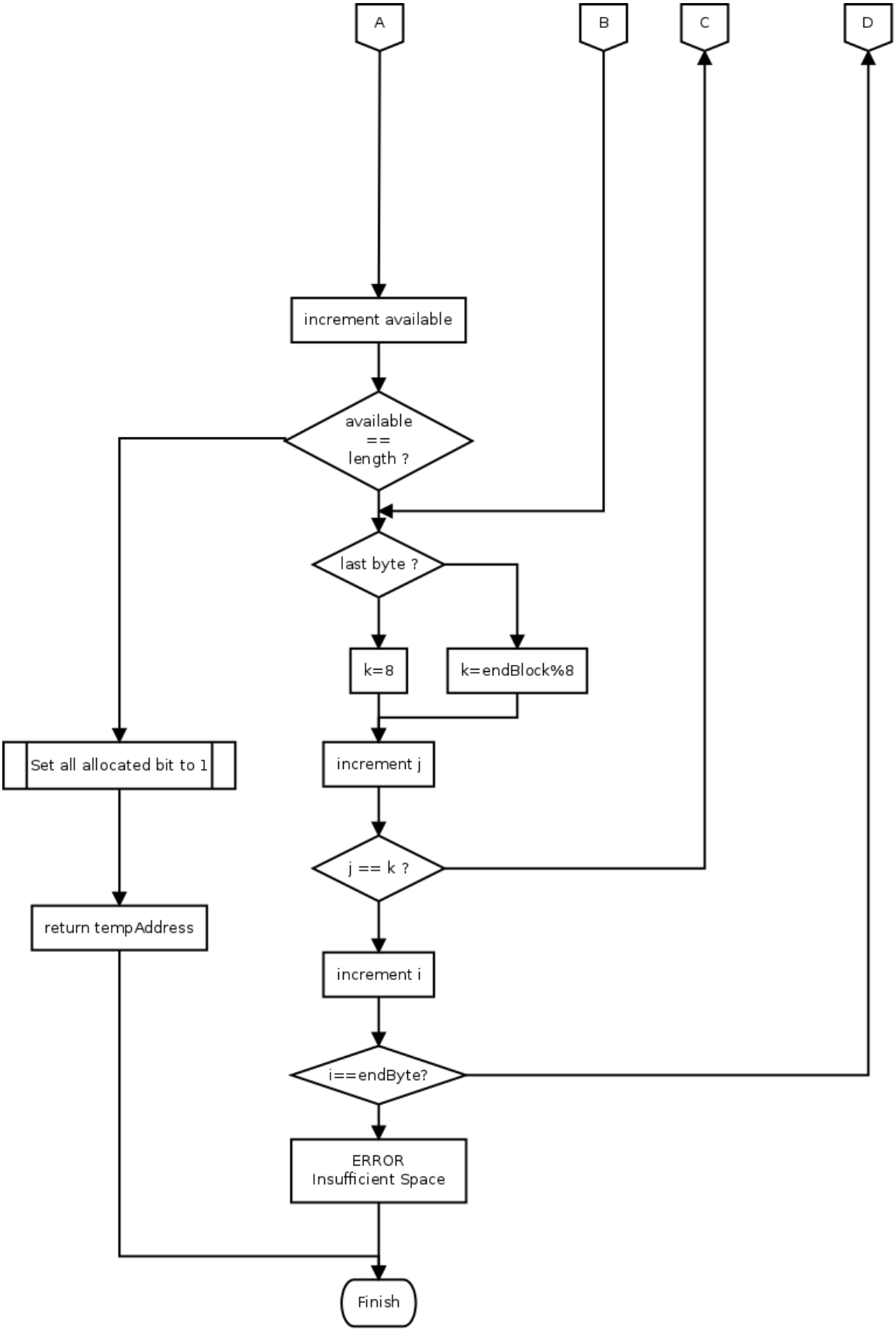

Figure 8.25: Algoritma fungsi Alloc Block(s)

- Begin Block = 1026
- End Block = 1037
- Length = 3

Block 1026 (begin block) ditunjuk oleh byte ke 256 (=1026/8) dan bit ke 2 (=1026%8) pada Block Allocation Table (BAT). Sementara Block 1037 (end block) ditunjuk oleh byte ke 257 (=1037/8) dan bit ke 5 (=1037%8). Karenanya, pencarian block yang bebas dimulai dari byte 256 (begin byte) bit ke-2 (begin bit) hingga byte 257 (end Byte) bit ke-5 (end Bit).

Pencarian akan menemukan bahwa 2 block pertama telah digunakan (warna merah) dan terdapat sejumlah block bebas (warna putih) yang cukup untuk memenuhi permintaan dimulai dari block 1028, yang ditunjuk oleh byte ke 256 dan bit ke-4 dari BAT. Fungsi Alloc Block(s) kemudian akan menandai ketiga block ini (warna hijau) sebagai block yang telah digunakan pada BAT, dan mengembalikan nilai 1028 sebagai awal block yang dialokasikan ke fungsi pemanggil.

Pengujian

Input			**Output**	
Begin	**End**	**Length**	**Address**	**Return Value**
0x0000	0x00ff	0x0f	0x0000	FS_OK
0x0000	0x000f	0xff	0x0000	FS_ERROR_INSUFFICIENT
0x00ff	0x01ff	0xff	0x00ff	FS_OK

Table 8.20: Test Vector Fungsi File System Alloc Block

Tabel 8.20 menampilkan Test Vector yang digunakan untuk menguji fungsi File System Alloc Block.

Implementasi

Tabel 8.21 menampilkan purwarupa dari implementasi fungsi File System Alloc Block.

Listing 8.9 menampilkan potongan program yang mengimplementasi fungsi FS Alloc.

```
1 int FSAlloc(uint16_t size, uint16_t startBlock, uint16_t endBlock,
     uint16_t * address)
2 {
3   uint16_t i, j, k, m, n;
```

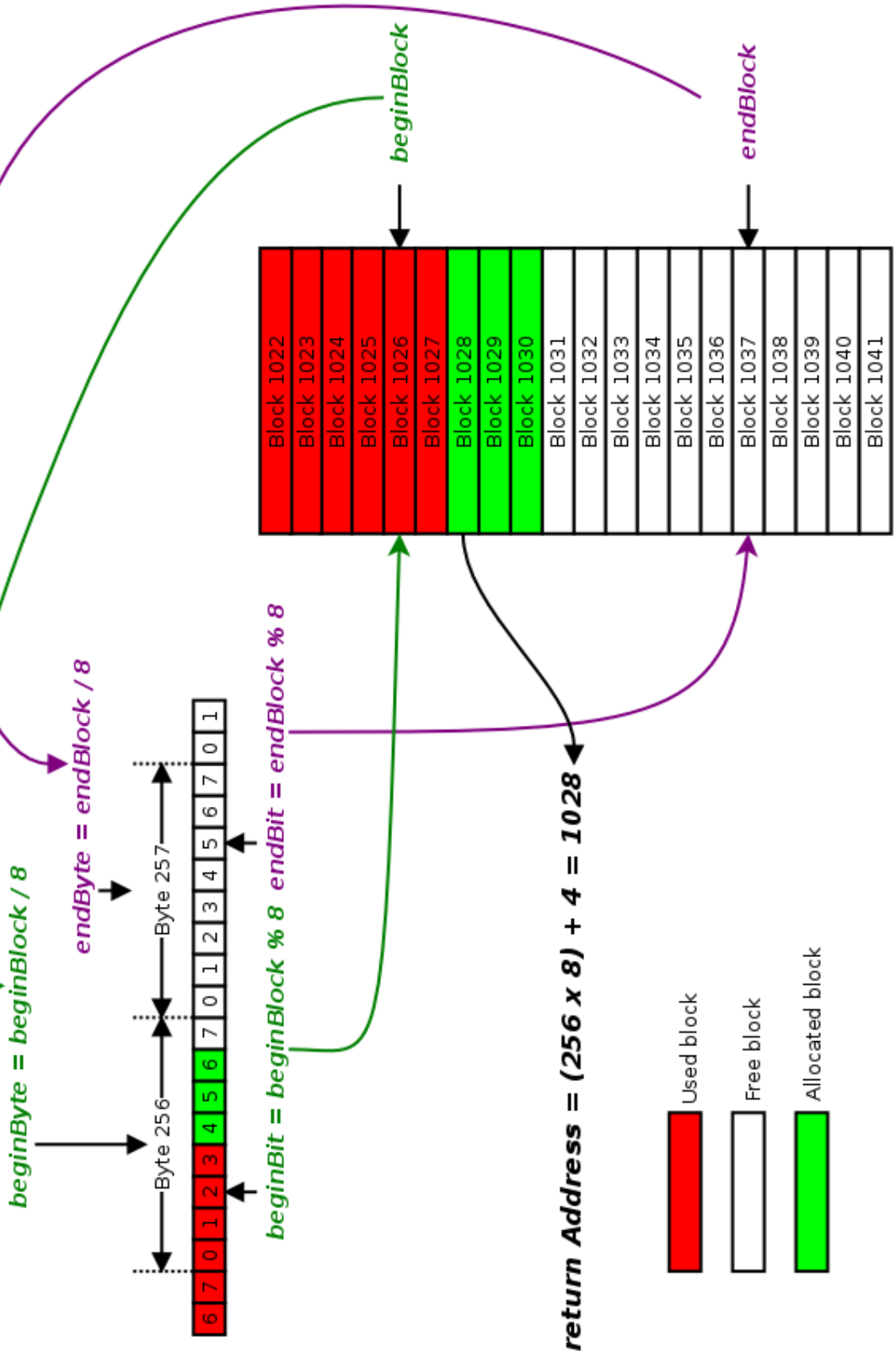

Figure 8.26: Contoh penggunaan Alloc Block(s)

Name	FSAlloc
Input	• Begin of Block to search for free blocks • End of Block to search for free blocks • Length of block to alloc
Output	• Address of allocated block • Status (if ERROR)

Table 8.21: Prototype Fungsi Alloc Block(s)

```
4  uint16_t startbyte, startbit, startSearchBit;
5  uint16_t stopbyte, stopbit, stopSearchBit;
6  uint16_t tempAddress;
7  uint16_t free = 0;
8  uint8_t temp = 0;
9  bool cont = 0;
10
11 for(i = (startBlock/8); i < (endBlock/8); i++) {
12
13   //read allocation table (byte -> 8 block)
14   temp = Memory_ReadByte(FS_START + FS_ALLOC_TABLE_OFFSET + i);
15
16   startSearchBit = (i == startBlock/8) ? startBlock%8 : 0;
17   stopSearchBit = (i == (endBlock/8)-1) ? endBlock%8 : 8;
18
19   // bit per bit operation
20   for(j = startSearchBit; j < stopSearchBit; j++) {
21
22     //check if MSB = 0
23     if((temp & 0x80) == 0) {
24
25       //check if this the first 0
26       if(cont == 0) {
27
28         //make a mark and save the address
29         cont = 1;
30         tempAddress = (i*8)+j;
```

```
31          startbyte = i;
32          startbit = j;
33
34        }
35
36        //increment free block
37        free ++;
38
39        //if free block is sufficient
40        if(free == size) {
41
42          stopbyte = i;
43          stopbit = j;
44
45          k = startbyte;
46          while(k <= stopbyte) {
47
48            m = 0;
49            n = 8;
50
51            if(k == startbyte) m = startbit;
52            if(k == stopbyte) n = stopbit;
53
54            for(m;m<=n;m++){
55              temp = Memory_ReadByte(FS_START + FS_ALLOC_TABLE_OFFSET
                      + k);
56              temp = temp | ((uint8_t)128>>m);
57              Memory_WriteByte(FS_START + FS_ALLOC_TABLE_OFFSET + k,
                    temp);
58            }
59
60            k++;
61
62          }
63
64          //return the first address of free block
65          *address = tempAddress;
66
67          return FS_OK;
68
69        }
70
71      }
```

```
72      else {
73
74        //if the MSB == 1 and continue
75        if(cont == 1) {
76
77          cont = 0;
78          free = 0;
79
80        }
81
82      }
83
84      temp = temp << 1;
85
86    }
87
88  }
89
90  return FS_ERROR_INSUFFICIENT_SPACE;
91 }
```

Listing 8.9: Listing Program Fungsi FS Alloc

8.2.11 Free Block(s)

Fungsi *Free Block(s)* merupakan kebalikan dari fungsi *Alloc Block(s)*, dimana sejumlah block yang telah dialokasikan (ditandai dengan nilai 1 pada bit yang bersesuaian di *block allocation table*) dibebaskan sehingga dapat dipergunakan kembali nantinya. Fungsi ini utamanya akan dipanggil oleh fungsi *Delete File*. Pembebasan dilakukan dengan mengubah nilai bit pada *block allocation table* menjadi 0 kembali.

Pengujian

Input		Output
Begin	**End**	**Return Value**
0x0000	0x00ff	FS_OK

Table 8.22: Test Vector Fungsi File System Free Block

Tabel 8.22 menampilkan Test Vector yang digunakan untuk menguji fungsi File System Free Block.

Implementasi

Tabel 8.23 menampilkan purwarupa dari implementasi fungsi File System Free Block.

Name	FSAlloc
Input	• Begin of block to free • Length of block to free
Output	• Status

Table 8.23: Prototype Fungsi Free Block(s)

Listing 8.10 menampilkan potongan program yang mengimplementasi fungsi FS Free.

```
1  int FSFree(uint16_t address, uint16_t length)
2  {
3    uint16_t startbyte, stopbyte;
4    uint8_t startbit, stopbit;
5    uint8_t startXOR, stopXOR;
6    uint8_t temp, tempXOR;
7    int i,j;
8
9    startbyte = address/8;
10   startbit = address%8;
11   stopbyte = (address+length)/8;
12   stopbit = (address+length)%8;
13
14   for(i = startbyte; i <= stopbyte; i++){
15     temp = Memory_ReadByte(FS_START + FS_ALLOC_TABLE_OFFSET + i);
16
17     startXOR = (i == startbyte) ? startbit : 0;
18     stopXOR = (i == stopbyte) ? stopbit : 8;
19
20     tempXOR = 0;
21
22     for(j = startXOR; j < stopXOR; j++)
```

```
23      tempXOR = tempXOR ^ (128 >> j);
24
25    temp = temp ^ tempXOR;
26
27    Mem_WriteByte(FS_START + FS_ALLOC_TABLE_OFFSET + i, temp);
28   }
29
30   return FS_OK;
31
32 }
```

Listing 8.10: Listing Program Fungsi FS Free