2. TEORI PENUNJANG

Pada bab ini yang akan dibahas mengenai teori – teori dasar yang diperlukan untuk mewujudkan tiap metode yang ada.

2.1. Pemrograman TCP/IP dengan Winsock

Sebelum kita membahas tentang winsock, ada baiknya kita bahas dulu sekilas tentang *socket* itu sendiri.

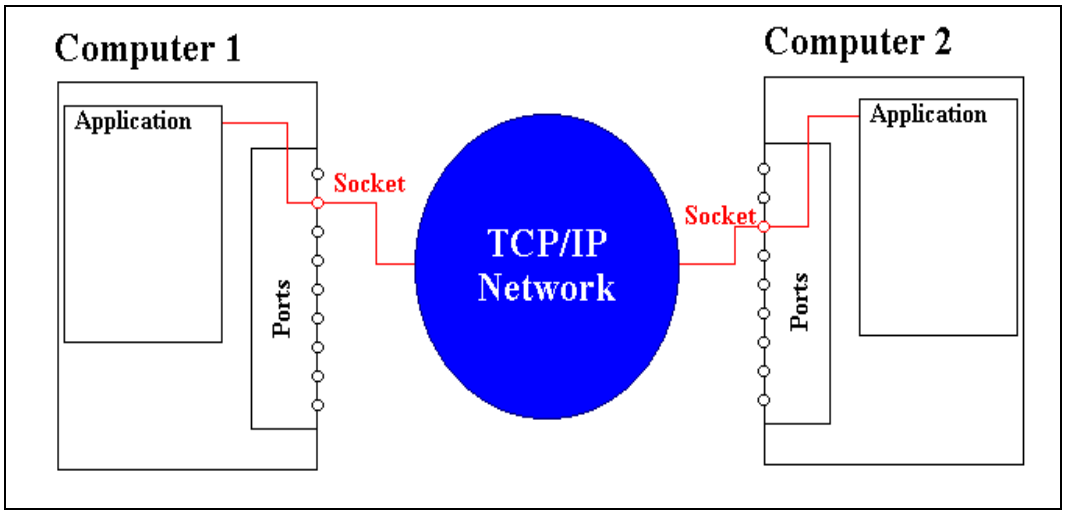
2.1.1. Sekilas tentang *Socket* dan Protokol

S*ocket* adalah sebuah penghubung antara 2 komputer dalam sebuah jaringan TCP/IP. *Socket* digunakan untuk bertukar berbagai macam informasi seperti file, *mail*, *website*, *multiplayer game* dan sebagainya.

Ada 2 macam tipe protokol, *Transmission Control Protocol* (TCP) dan *User Datagram Protocol* (UDP). Protokol adalah seperangkat aturan yang digunakan untuk menentukan dengan cara apa informasi yang ada dikirim dan diterima dalam sebuah jaringan TCP/IP. Protokol-protokol yang lain seringkali digunakan di atas TCP dan UDP, misalnya *File Transfer Protocol* (FTP), sebuah protokol untuk pertukaran file yang menggunakan TCP sebagai penghubung dengan *file server*. TCP dan UDP sendiri dibangun diatas IP (*Internet Protocol*).

Sebuah *socket* TCP bisa menjamin bahwa informasi yang dikirimkan sampai di tujuan dengan baik, sedangkan *socket* UDP tidak bisa menjamin sampai atau tidaknya informasi yang dikirimkan ke tujuan sebelum pihak penerima memberikan balasan.

Untuk menghubungkan *socket* pada sebuah komputer dengan *socket* pada komputer lainnya, perlu untuk mengetahui IP-*address* komputer yang dituju beserta *port* yang akan dituju. Tiap komputer memiliki 65536 *port*. Beberapa *port* biasanya telah digunakan oleh beberapa protokol, misalnya *port* 80 biasanya digunakan untuk HTTP (*HyperText Transfer Protocol*), *port* 21 digunakan untuk FTP (*File Transfer Protocol*).



Gambar 2.1. Ilustrasi Hubungan antar *Socket* pada 2 Komputer

Sumber: *An Introduction to TCP/IP Sockets and Winsock*. 23 January 2004 <http://frenchwhale.gotdns.com/winsockt/Lesson1.htm>

2.1.2. Sekilas tentang Winsock

Winsock (Winsock 2) merupakan standar untuk pemrograman jaringan pada sistem operasi Windows. Winsock 2 diciptakan sebagai antarmuka pemrograman TCP/IP pada semua versi sistem operasi Windows mulai dari Windows 3.x, Windows 95/98, Windows NT, Windows 2000, Windows XP bahkan untuk Windows CE.

Winsock 2 memiliki beberapa keuntungan, antara lain:

- Winsock 2 mampu bekerja pada beberapa protokol, tidak hanya pada protokol TCP/IP saja, karena winsock 2 memiliki fungsi-fungsi baru yang dibutuhkan untuk menangani beberapa protokol. Kemampuan menangani beberapa protokol ini lebih menguntungkan karena pada Winsock sebelumnya hanya mampu menangani protokol TCP/IP saja.
- Winsock 2 menyediakan sebuah standar antarmuka yang disebut SPI yang menghubungkan antara DLL winsock (WS2_32.dll) dengan *protocol stack*. Dengan demikian, WS2_32.dll dapat mengakses beberapa *protocol stack* dari beberapa perusahaan yang berbeda secara simultan. Tidak seperti winsock 1.1 yang mengharuskan tiap-tiap perusahaan pembuat *protocol stack* mnyediakan *library* untuk mengimplementasikan antarmuka winsock. Tentu saja, *library* yang dibuat oleh suatu perusahaan berbeda dengan *library* yang dibuat oleh

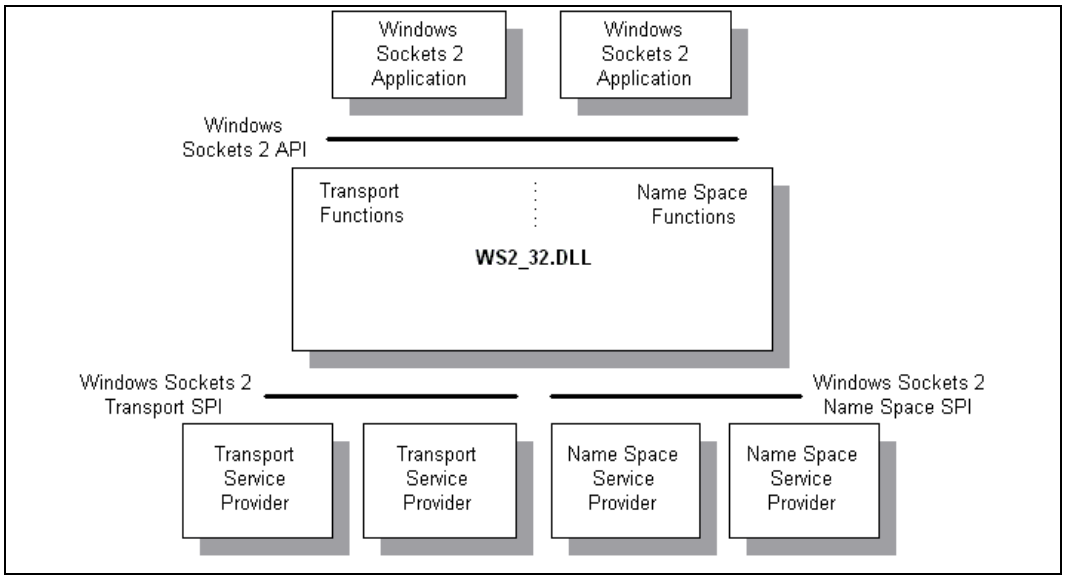
perusahaan lainnya, dan ini akan sangat menyulitkan seorang pemrogram bila harus menulis ulang aplikasinya agar dapat berjalan pada protokol yang lain.

- Kode sumber aplikasi yang ditulis dengan winsock 1.1 dapat dengan mudah disesuaikan dengan sistem winsock 2 tanpa melakukan modifikasi terlalu banyak pada kode sumber.

2.1.3. Arsitektur Winsock

Arsitektur winsock 1.1 membatasi hanya satu winsock.dll yang aktif dalam sebuah sistem. Hal ini menyebabkan, sistem akan kesulitan untuk mengimplementasikan lebih dari satu winsock dalam waktu yang bersamaan.

Winsock 2 memiliki arsitektur yang lebih fleksibel sehingga mampu mendukung *multiple protocol stack*, antarmuka dan *service provider* secara simultan. Pada bagian *layer* atas winsock 2 memang masih menggunakan satu DLL (winsock 2 API), namun terdapat *layer* lagi di bawahnya dan sebuah *service provider interface* standar, yang mana masing-masing juga bersifat fleksibel.



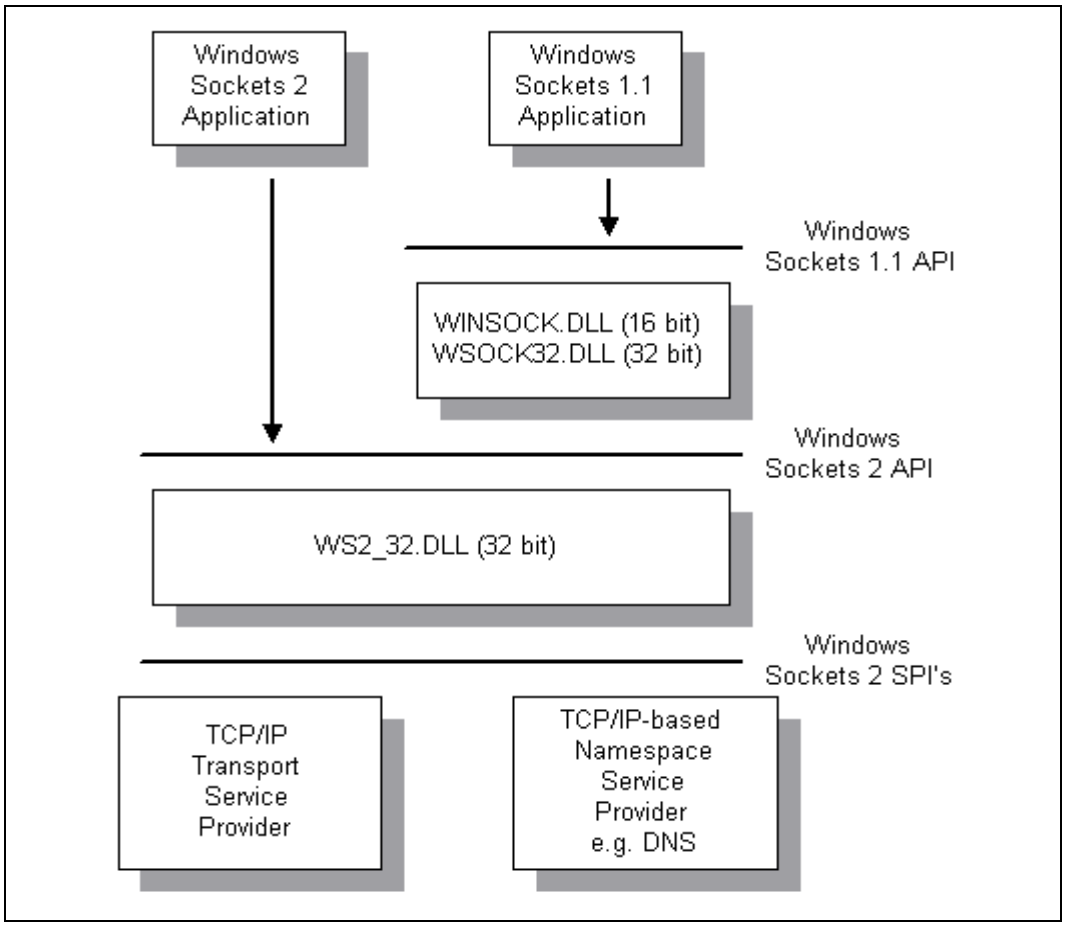
Gambar 2.2. Arsitektur Winsock 2

Sumber: *Windows Sockets 2 Architecture*. 23 January 2004 <ms-help://MS.MSDNQTR.2002APR.1033/winsock/ovrvw1_5kky.htm>

Winsock 2 mengadopsi model Windows *Open Systems Architecture* yang memisahkan API dari *Protocol Service Provider*. Dengan menggunakan model

ini, winsock menyediakan standar API, dan tiap-tiap *vendor* menyediakan layer *service provider*-nya sendiri. Layer API berhubungan dengan layer *service provider* melalui sebuah standar *Service Provider Interface* (SPI), yang mana SPI tersebut mampu menghubungkan layer API dengan layer *service provider* dari berbagai *vendor*.

Beberapa fasilitas baru winsock 2 membutuhkan banyak perubahan pada arsitekturnya. Meskipun arsitektur winsock 2 jauh berbeda dengan arsitektur winsock 1.1, winsock 2 masih mendukung aplikasi-aplikasi yang ditulis dengan winsock versi sebelumnya. Namun demikian, penanganan WS2_32.dll untuk aplikasi-aplikasi yang ditulis dengan winsock 1.1 baik yang 16-bit maupun 32-bit, memiliki sedikit perbedaan dengan aplikasi yang ditulis dengan winsock 2.

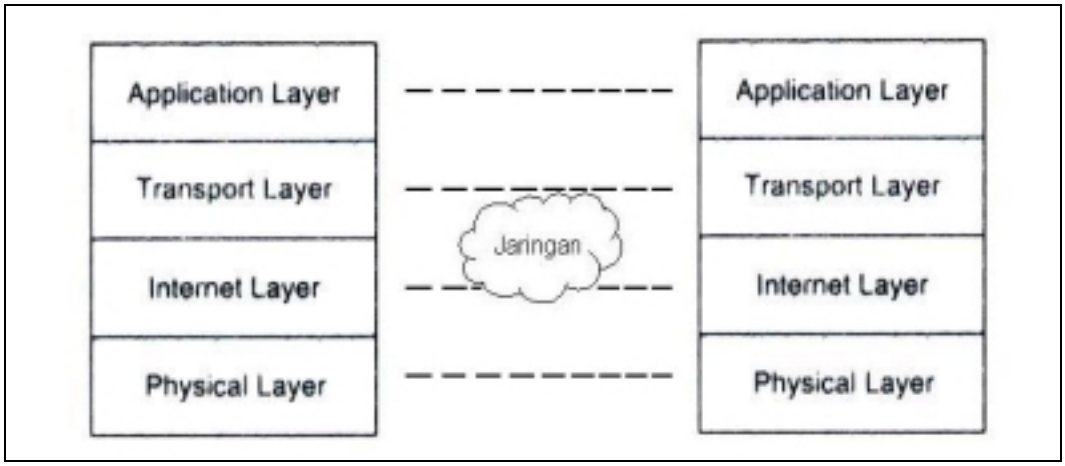


Gambar 2.3. Perbedaan Alur dari Aplikasi Winsock 2 dan Winsock 1.1 yang ditangani oleh WS2_32.dll

Sumber: *Changes from Windows Sockets 1.1*. 23 January 2004 <ms-help://MS.MSDNQTR.2002APR.1033/winsock/ovrvw1_2xv6.htm>

2.1.4. Model Protokol TCP/IP

Protokol TCP/IP merupakan hasil pengembangan dan riset protokol yang dilakukan atas jaringan paket-*switched* eksperimental, ARPANET, dan mendapat dukungan dana dari DARPA (*Defense Advanced Research Project Agency*). Secara umum, TCP/IP ini lebih dikenal sebagai TCP/IP *protocol suite*.



Gambar 2.4. Model Protokol TCP/IP

Sumber: Stallings, W. *Data and Computer Communications 6th Edition*. New Jersey: Prentice-Hall, Inc, 2000, p. 19

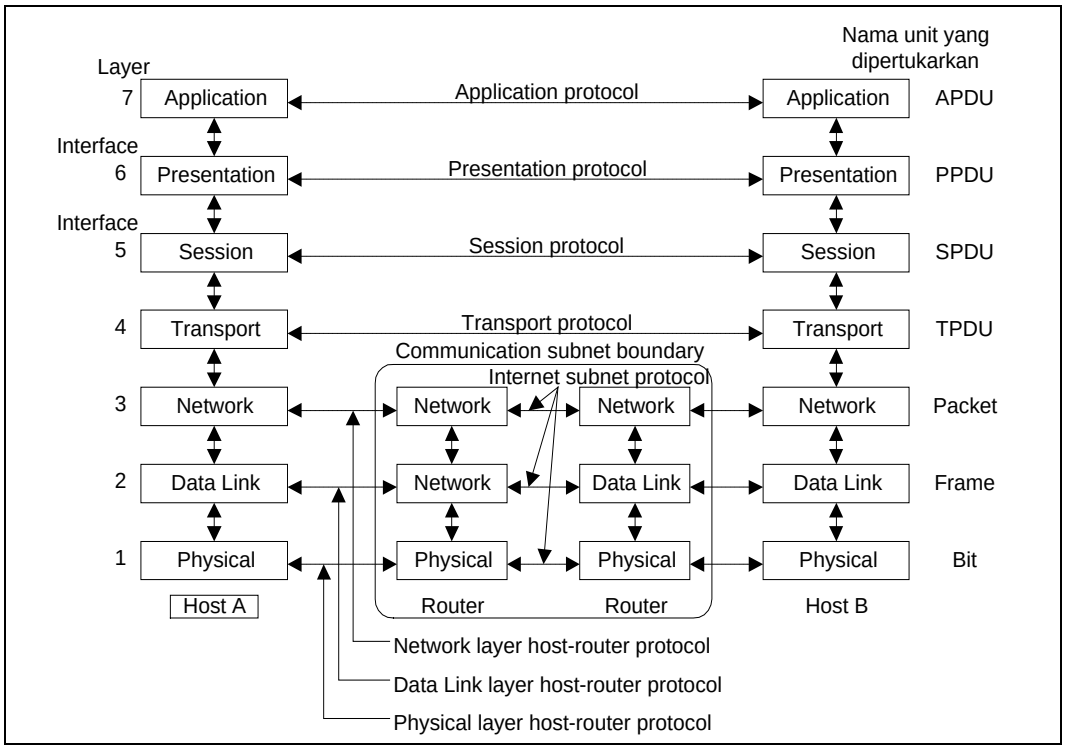
Model protokol TCP/IP terdiri dari empat *layer*, yaitu:

- ***Application layer*** berisi logika yang dibutuhkan untuk mendukung aplikasi. Untuk jenis aplikasi yang berbeda diperlukan modul-modul terpisah yang sesuai dengan aplikasi tersebut.
- ***Transport layer*** berfungsi memastikan seluruh data yang sampai di tujuan sesuai dengan data yang dikirim oleh pengirim. Oleh karena itu, *layer* ini harus memiliki mekanisme yang handal supaya data yang sampai di tujuan tidak ada yang hilang atau rusak.
- ***Internet layer*** berfungsi mengatur mekanisme pertukaran data antara dua ujung sistem yang dihubungkan ke jaringan yang berbeda. IP (*Internet Protocol*) dipergunakan pada *layer* ini untuk menyediakan fungsi *routing* melintasi jaringan yang bermacam-macam. Protokol ini diterapkan tidak hanya pada ujung-ujung sistem , namun juga pada jalur-jalurnya.

- ***Physical layer*** meliputi antarmuka fisik antara suatu perangkat transmisi data (komputer, *workstation*) dengan sebuah media transmisi atau jaringan. *Layer* ini berkaitan dengan karakteristik khusus dari media transmisi, sifat sinyal, data *rate*, dan hal-hal lain yang berkaitan dengan hal itu.

2.1.5. Model OSI

Model OSI (*Open Systems Interconnection*) dikembangkan sebagai model untuk arsitektur komunikasi komputer, serta sebagai kerangka kerja bagi pengembangan standar-standar protokol. Model OSI menggambarkan bagaimana informasi dari suatu aplikasi pada sebuah komputer berpindah melewati sebuah media jaringan ke suatu aplikasi di komputer lain. Model ini ditujukan bagi pengkoneksian *open system*, karena model OSI ini merupakan sistem yang terbuka untuk melakukan komunikasi dengan sistem-sistem Model OSI secara konseptual terdiri dari tujuh *layer*, dimana masing-masing *layer* memiliki fungsi yang khusus, seperti yang dijelaskan pada gambar 2.5.



Gambar 2.5. Model Referensi OSI

Sumber: Stallings, W. *Data and Computer Communications 6th Edition*. New Jersey: Prentice-Hall, Inc, 2000, p. 52.

Tujuh *layer* pada model OSI dapat dibagi menjadi dua bagian, *layer* atas dan *layer* bawah. *Layer* atas berhubungan dengan persoalan aplikasi dan biasanya diimplementasikan pada *software*. *Layer-layer* yang termasuk *layer* atas adalah *application layer*, *presentation layer* dan *session layer*. Sedangkan *layer* bawah berhubungan dengan persoalan *transport* data. *Layer-layer* yang termasuk *layer* bawah adalah *transport layer*, *network layer*, *data-link layer* dan *physical layer*. *Physical layer* dan *data-link layer* diimplementasikan dalam *hardware* dan *software*. *Layer-layer* bawah yang lain umumnya hanya diimplementasikan dalam *software*. Berikut ini penjelasan fungsi-fungsi khusus yang ada pada setiap *layer*:

- ***Application layer*** menyediakan suatu cara bagi program-program aplikasi untuk mengakses lingkungan OSI. *Layer* ini berisi fungsi-fungsi manajemen dan mekanisme-mekanisme yang umumnya berguna untuk mendukung aplikasi-aplikasi yang didistribusikan. Selain itu, aplikasi dengan tujuan penggunaan umum seperti file transfer, email dan terminal *access* untuk komputer-komputer yang berjauhan ditempatkan pada *layer* ini.
- ***Presentation layer*** menentukan format data yang dipindahkan antar aplikasi dan menawarkan pada program-program aplikasi serangkaian layanan transformasi data. *Layer* ini menentukan sintaks yang dipergunakan antar aplikasi serta menyediakan modifikasi seleksi dan *subsequent* dari representasi yang dipergunakan. Contoh layanan khusus pada *layer* ini adalah kompresi dan enkripsi data.
- ***Session layer*** menyediakan mekanisme untuk mengatur dialog antar aplikasi pada ujung-ujung sistem. Pada beberapa kasus, akan ada sedikit atau bahkan sama sekali tidak diperlukan layanan dari *layer* ini, namun untuk beberapa aplikasi tertentu, layanan pada *layer* ini tetap diperlukan.
- ***Transport layer*** meyediakan suatu mekanisme perubahan data di antara ujung-ujung sistem. Layanan *transport* pada *connection-oriented* menjamin bahwa data yang dikirim bebas dari kesalahan, secara bertahap, dengan tidak mengalami duplikasi atau hilang. *Layer* ini juga dapat dikaitkan dengan mengoptimalisasikan penggunaan layanan jaringan dan menyediakan mutu layanan yang bisa diminta untuk entiti sesi. Sebagai contoh, entiti sesi bisa menentukan laju *error* yang boleh diterima, maksimum penundaan, prioritas

dan pengamanan. Ukuran dan kelengkapan sebuah protokol *transport* tergantung dari seberapa dapat diandalkan atau tidak jaringan yang mendasari dan layanan *network layer*. Karena itu, ISO telah mengembangkan suatu rumpun lima standar protokol *transport* yang masing-masing berorientasi terhadap layanan yang berbeda yang mendasari. Pada protokol TCP/IP, terdapat dua protokol *transport layer* yang umum: TCP yang bersifat *connection-oriented* dan UDP yang bersifat *connectionless*.

- ***Network layer*** menyediakan transfer informasi diantara ujung-ujung sistem melewati beberapa jaringan komunikasi berurutan. Ini membantu mengurangi beban pada *layer* tertinggi dari kebutuhan untuk mengetahui apapun mengenai transmisi data yang mendasari dan mengganti teknologi yang dipergunakan untuk menghubungkan sistem. Pada *layer* ini, sistem komputer berdialog dengan jaringan untuk menentukan alamat tujuan dan meminta fasilitas jaringan tertentu, misalnya prioritas.

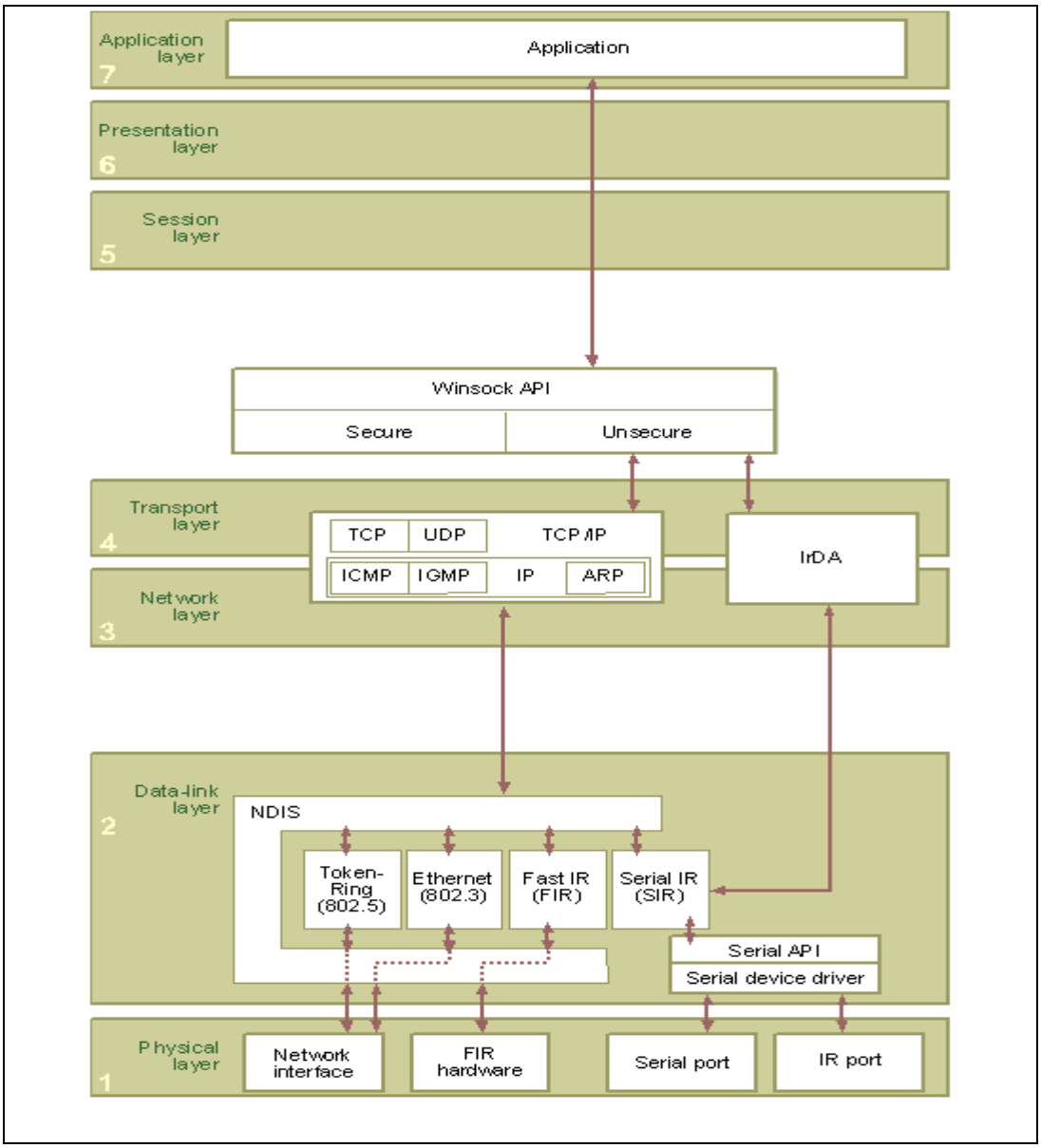
  Namun ada kemungkinan untuk menghalangi fasilitas-fasilitas komunikasi yang dikelola oleh *network layer*, misalnya, adanya *link* langsung dari titik ke titik diantara ujung sistem. Dalam hal ini, tidak diperlukan lagi *network layer* karena *data-link layer* mamppu menyediakan fungsi yang diperlukan dalam mengelola *link*.
- ***Data-Link layer*** mengupayakan agar *link* fisik cukup baik dan menyediakan alat-alat untuk mengaktifkan, mempertahankan dan menonaktifkan *link*. Layanan utama yang disediakan oleh *data-link layer* untuk *layer* yang lebih tinggi adalah *error detection* dan kontrol. Jadi, jika *layer* ini berfungsi dengan baik, maka *layer* diatasnya bisa menerima transmisi bebas kesalahan melewati *link*.
- ***Physical layer*** mencakup *physical interface* diantara divais-divais dan aturan bit-bit dilewatkan dari satu ke yang lain. *Physical layer* memiliki empat karakteristik penting, yaitu:
  - Mekanis: berkaitan dengan properti fisik dari *interface* ke media transmisi. Biasanya, spesifikasinya adalah dari konektor *pluggable* yang menggabungkan satu atau lebih *signal conductor*, yang disebut sirkuit.
  - Elektris: berkaitan dengan tampilan bit-bit.

- Fungsional: menentukan fungsi-fungsi yang ditampilkan oelh sirkuit tunggal dari *physical interface* diantara sebuah sistem dengan media transmisi.
- Prosedural: menentukan rangkaian kejadian dimana arus bit berpindah melalui medium fisik.

2.1.6. Hubungan antara Winsock dan Model OSI

Dalam organisasi internasional untuk standarisasi model *Open Systems Interconnection* (ISO/OSI), winsock beroperasi pada antarmuka *session layer* sampai *transport layer*. Winsock merupakan antarmuka antara aplikasi dengan *transport protocol* dan bekerja sebagai aliran data I/O.

Winsock menyederhanakan pengembangan aplikasi dalam *upper* ISO/OSI *layer* dengan menangani pertukaran data jaringan secara detail pada *lower layer*. Winsock menyediakan antarmuka yang dapat diprogram antara *upper layer* dan *lower layer*. Aplikasi winsock sendiri berada pada *upper layer* (*application layer*, *presentation layer* dan *session layer*). Pada aplikasi winsock, data dipaketkan dan dikirim melalui jaringan pada *lower layer* (*transport layer*, *network layer*, *data-link layer* dan *physical layer*). Gambar 2.6 menjelaskan hubungan antara winsock dengan model OSI.



Gambar 2.6. Hubungan antara Winsock dan Model OSI

Sumber: *Winsock and the OSI Model*. 23 January 2004 <ms-help://MS.MSDNQTR.2002APR.1033/wcecomm/htm/winsock_1.htm>

2.1.7. Fungsi-fungsi dalam Winsock

Setelah mengetahui tentang winsock dan arsitekturnya, sekarang akan dijelaskan fungsi-fungsi dalam winsock yang tentunya sangat perlu untuk diketahui agar bisa membangun sebuah aplikasi *client* / *server*. Tidak semua fungsi akan dibahas, melainkan hanya fungsi-fungsi dasar untuk pemrograman aplikasi.

2.1.7.1. Fungsi *accept*

Fungsi *accept* ini mempunyai tugas menerima koneksi yang berusaha masuk ke *socket*. Sintaknya sebagai berikut:

```
SOCKET accept(
      SOKCET s,
      struct sockaddr FAR *addr,
      int FAR *addrlen
      );
```

Parameter:

s [in] menunjuk pada *socket* yang sedang mendengarkan koneksi setelah pemanggilan fungsi *listen*.

addr [out] *optional pointer* ke *buffer* yang menerima alamat dari koneksi yang masuk , sebagai *communications layer*. Format *addr* ditentukan dengan *address family* yang dibuat ketika *socket* dari struktur SOCKADDR terbuat.

addrlen [out] *optional pointer* ke integer yang berisi panjang dari *addr*.

Jika tidak terjadi kesalahan, maka *accept* akan mengembalikan nilai yang bertipe *SOCKET* yang merupakan deskripsi untuk *socket* yang baru. Jika ada kesalahan, maka fungsi ini akan mengembalikan nilai INVALID_SOCKET dan keterangan tentang kesalahan yang terjadi dapat diperoleh dengan memanggil fungsi *WSAGetLastError*.

Fungsi *accept* memproses koneksi pertama dalam antrian pada *socket* ***s***. Fungsi *accept* kemudian menciptakan sebuah *socket* baru dan mengembalikan sebuah *handle* untuk *socket* tersebut. *Socket* baru inilah yang kemudian akan menangani koneksi yang sesungguhnya, dan *socket* ini memiliki property yang sama dengan *socket* ***s***. Fungsi *accept* hanya digunakan pada koneksi jenis *connection-oriented* dengan *socket* ***s*** bertipe SOCK_STREAM.

2.1.7.2. Fungsi *bind*

Fungsi *bind* mempunyai tugas mengikatkan *socket* yang telah dibuat pada sebuah *local address* tertentu. Sintaknya sebagai berikut:

```
int bind(
      SOCKET s,
      const struct sockaddr FAR *name,
      int namelen
      );
```

Parameter:

s [in] menunjuk pada *socket* yang belum diikatkan pada *local address*.

name [in] alamat dari struktur SOCKADDR dimana *socket* akan ditempatkan.

namelen [in] panjang nilai dari parameter *name*.

Jika tidak terjadi kesalahan, maka fungsi *bind* akan mengembalikan nilai nol. Sebaliknya jika terjadi kesalahan, maka fungsi ini akan mengembalikan nilai SOCKET_ERROR. Keterangan tentang kesalahan yang terjadi dapat diperoleh dengan memanggil fungsi *WSAGetLastError*.

Fungsi *bind* digunakan pada *socket* yang belum terkoneksi sebelum melakukan pemanggilan *connect* dan fungsi *listen*. Fungsi ini dipakai untuk mengikat semua jenis koneksi *socket*, baik koneksi *connection-oriented* ataupun *connectionless*.

2.1.7.3. Fungsi *closesocket*

Fungsi *closesocket* berfungsi untuk menutup keberadaan suatu *socket*. Sintaknya sebagai berikut:

```
int closesocket(
      SOCKET s
      );
```

Parameter:

s [in] menunjuk pada *socket* yang akan ditutup.

Jika tidak ada kesalahan, maka fungsi *closesocket* akan mengembalikan nilai nol. Tetapi jika terjadi kesalahan, maka fungsi ini akan mengembalikan nilai SOCKET_ERROR dan keterangan tentang kesalahan yang terjadi dapat dilihat dengan memanggil fungsi *WSAGetLastError*.

2.1.7.4. Fungsi *connect*

Fungsi *connect* berfungsi membuat koneksi dengan spesifikasi tertentu. Sintaknya sebagai berikut:

```
int connect(
      SOCKET s,
      const struct sockaddr FAR *name,
      int namelen
      );
```

Parameter:

s [in] menunjuk pada *socket* yang belum melakukan koneksi.

name [in] nama *socket* dalam struktur SOCKADDR yang mana merupakan tempat koneksi akan dibuat.

namelen [in] panjang nilai dari parameter *name*.

Jika tidak terjadi kesalahan, maka fungsi *connect* akan mengembalikan nilai nol. Sebaliknya jika terjadi kesalahan, maka fungsi ini akan mengembalikan nilai SOCKET_ERROR. Keterangan tentang kesalahan yang terjadi dapat diperoleh dengan memanggil fungsi *WSAGetLastError*.

Fungsi *connect* digunakan untuk menciptakan koneksi ke sebuah tujuan tertentu. Ketika koneksi sudah terbentuk, maka *socket* ***s*** siap untuk mengirim dan menerima data melalui koneksi yang terbentuk.

2.1.7.5. Fungsi *listen*

Fungsi *listen* berfungsi menempatkan *socket* pada suatu keadaan supaya bisa mendengarkan koneksi-koneksi yang datang. Sintaknya sebagai berikut:

```
int listen(
      SOCKET s,
      int backlog
      );
```

Parameter:

s [in] menunjuk pada *unbound socket*.

backlog [in] menyatakan panjang maksimum antrian koneksi.

Jika tidak terjadi kesalahan, maka fungsi *listen* akan menghasilkan nilai nol. Sebaliknya jika terjadi kesalahan, maka fungsi ini akan mengembalikan nilai SOCKET_ERROR dan untuk mendapatkan keterangan tentang kesalahannya dapat menggunakan fungsi *WSAGetLastError*.

Untuk menerima koneksi, sebuah *socket* harus terlebih dahulu diciptakan dengan menggunakan fungsi *socket* dan diikat pada suatu alamat dengan menggunakan fungsi *bind*. Kemudian fungsi *listen* bertugas mendengar koneksi yang datang pada *socket* dan fungsi *accept* yang akan menerima koneksi yang datang tersebut. Jenis koneksi yang menggunakan fungsi *listen* untuk mendengarkan koneksi dari *client* biasanya merupakan *socket* yang diciptakan

oleh fungsi *socket* dengan tipe SOCK_STREAM dan bersifat *connection-oriented*.

2.1.7.6. Fungsi *recv*

Fungsi *recv* berfungsi untuk menerima data dari *socket* yang terhubung. Sintaknya sebagai berikut:

```
int recv(
      SOCKET s,
      char FAR *buf,
      int len,
      int flags
      );
```

Parameter:

s [in] menunjuk pada *socket* yang terkoneksi.

buf [out] *buffer* untuk data yang diterima.

len [in] panjang dari *buffer*.

flags [in] merupakan cara pemanggilan dibuat.

Jika tidak terjadi kesalahan, maka fungsi ini akan mengembalikan jumlah *byte* yang diterima dan akan mengembalikan nilai SOCKET_ERROR jika terjadi kesalahan. Keterangan tentang kesalahan yang terjadi dapat diperoleh dengan pemanggilan fungsi *WSAGetLastError*. Jika koneksi terputus, maka fungsi ini akan mengembalikan nilai nol.

Fungsi *recv* digunakan untuk membaca data yang masuk pada *socket* baik yang terkoneksi dengan protokol *connecion-oriented* ataupun *connectionless*. Ketika menggunakan protokol *connecion-oriented,* sebuah *socket* harus memiliki koneksi dulu sebelum memanggil fungsi *recv*. Sedangkan jika menggunakan protokol *connectionless*, sebuah *socket* harus diikat ke suatu alamat tertentu sebelum memanggil fungsi *recv*.

Untuk *socket connection-oriented* pemanggilan fungsi *recv* akan mengembalikan sejumlah informasi yang tersedia sebatas ukuran *buffer* yang telah ditentukan. Jika sisi *remote* telah menghentikan koneksi, dan semua data telah diterima, fungsi *recv* dengan segera akan melengkapinya dengan 0 *byte*.

2.1.7.7. Fungsi *send*

Fungsi *send* berfungsi untuk mengirimkan data ke *socket* yang terhubung. Sintaknya sebagai berikut:

```
int send(
      SOCKET s,
      const char FAR *buf,
      int len,
      int flags
      );
```

Parameter:

s [in] menunjuk pada *socket* yang terkoneksi.

buf [out] *buffer* berisi data yang akan dikirimkan.

len [in] panjang dari *buffer*.

flags [in] merupakan cara pemanggilan dibuat.

Jika tidak terjadi kesalahan pada waktu pemanggilan fungsi ini, maka fungsi ini akan mengembalikan jumlah *byte* yang telah dikirimkan, yang mana besarnya dapat kurang dari angka yang disebutkan pada parameter ***len*** untuk model *socket nonblocking*. Sebaliknya, jika terjadi kesalahan maka fungsi ini akan mengembalikan nilai SOCKET_ERROR, dan kode kesalahan yang terjadi dapat diketahui dengan memanggil fungsi *WSAGetLastError*.

Pemanggilan fungsi *send* dengan parameter ***len*** berisi nilai nol diperbolehkan. Dalam banyak kasus, fungsi *send* akan mengembalikan nilai nol sebagai nilai yang sah.

2.1.7.8. Fungsi *shutdown*

Fungsi *shutdown* berfungsi untuk mematikan kemampuan sebuah *socket* untuk mengirim dan menerima data. Sintaknya sebagai berikut:

```
int shutdown(
      SOCKET s,
      int how
      );
```

Parameter:

s [in] menunjuk pada sebuah *socket*.

how [in] menunjuk pada tipe operasi apa saja yang tidak lagi dapat dijalankan.

Jika tidak terjadi kesalahan, maka fungsi *shutdown* akan mengembalikan nilai nol. Jika terjadi kesalahan, maka fungsi ini akan mengembalikan nilai SOCKET_ERROR, dan keterangan tentang kesalahan yang terjadi dapat diketahui dengan pemanggilan fungsi *WSAGetLastError*.

Fungsi *shutdown* digunakan oleh semua tipe *socket* untuk mematikan kemampuan menerima dan mengirimkan data. Jika parameter ***how*** adalah SD_RECEIVE, akan mengakibatkan pemanggilan fungsi *recv* pada *socket* tidak lagi diijinkan. Hal ini tidak mempengaruhi *layer-layer* protokol dibawahnya.

Jika parameter ***how*** adalah SD_SEND, akan mengakibatkan pemanggilan fungsi *send* tidak lagi diperbolehkan. Untuk *socket* TCP, FIN akan dikirimkan setelah semua data dikirim dan menerima pemberitahuan dari penerima. Sedangkan pemberian nilai SD_BOTH ke parameter ***how*** akan melumpuhkan fungsi *recv* dan fungsi *send* secara bersamaan.

Pemanggilan fungsi ini tidak akan menutup *socket*. Penutupan *socket* akan dilakukan jika terjadi pemanggilan fungsi *closesocket*. Untuk lebih meyakinkan bahwa semua data telah dikirim dan diterima pada *socket* yang terhubung sebelum *socket* tersebut ditutup, sebuah aplikasi haruslah memanggil fungsi *shutdown* untuk menutup koneksi sebelum benar-benar menutup *socket* dengan memanggil fungsi *closesocket*.

Meskipun pemanggilan fungsi *shutdown* tidak menutup *socket* yang terhubung, namun tidak dianjurkan untuk menggunakan kembali *socket* tersebut. Pada umumnya, winsock tidak mendukung pemakaian fungsi *connect* pada *socket* yang telah melakukan pemanggilan fungsi *shutdown*.

2.1.7.9. Fungsi *socket*

Fungsi *socket* digunakan untuk membuat sebuah *socket* yang akan diikatkan pada *service provider* tertentu. Sintaknya sebagai berikut:

```
SOCKET socket(
      int af,
      int type,
      int protocol
      );
```

Parameter:

af [in] alamat *family* yang telah ditentukan.

type [in] tipe untuk *socket* baru. Tipe *socket* ada 2 macam, SOCK_STREAM dan SOCK_DGRAM. Tipe SOCK_STREAM umumnya digunakan untuk koneksi TCP, dan tipe SOCK_DGRAM untuk koneksi UDP.

protocol [in] digunakan dengan *socket* yang secara khusus untuk menunjukkan alamat *family*.

Jika tidak terjadi kesalahan, maka fungsi *socket* akan mengembalikan sebuah referensi yang menunjuk ke *socket* baru yang telah dibuat. Jika fungsi *socket* gagal membuat sebuah *socket* baru, maka fungsi ini akan mengembalikan nilai INVALID_SOCKET. Keterangan tentang kesalahan yang terjadi pada fungsi ini akan diperoleh dengan memanggil fungsi *WSAGetLastError*.

Tipe *socket* SOCK_STREAM menyediakan koneksi *full-duplex*, dan tiap *socket* harus benar-benar terhubung sebelum proses pertukaran data dilakukan. Koneksi ke *socket* yang lain dibuat dengan memanggil fungsi *connect*. Setelah terhubung, data dapat ditransfer dengan menggunakan fungsi *send* dan fungsi *recv*. Untuk mengakhiri koneksi dapat dilakukan dengan memanggil fungsi *closesocket*.

2.2. Video *Capture*

Untuk keperluan pengambilan gambar dari kamera, windows telah menyediakan kelas AVICap. AVICap menyediakan fungsi dan *message* untuk mengakses perangkat keras video dan audio serta mengatur proses *streaming* video *capture* ke *disk*.

2.2.1. Sekilas tentang AVICap

AVICap mendukung *streaming* video *capture* dan *single-frame capture* secara *real-time*. AVICap juga menyediakan kontrol untuk sumber video yaitu *Media Control Interface* (MCI) sehingga pengguna dapat mengkontrol posisi awal dan akhir dari sumber video melalui aplikasi.

Dengan kelas AVICap, kita dapat melakukan hal-hal berikut:

- Mengambil *stream* audio dan video untuk dijadikan bentuk file audio-video *interleaved* (AVI).
- Menghubungkan dan menghentikan hubungan dengan peralatan audio-video secara dinamis.
- Menampilkan secara langsung sinyal video yang diterima dari peralatan seperti kamera dengan menggunakan *method overlay* atau *preview*.
- Menentukan besar *capture rate*.
- Menampilkan kotak dialog yang berisi kontrol untuk format video.
- Membuat, menyimpan dan memanggil kembali *pallete*.
- Menyalin gambar dan *pallete* ke dalam *clipboard*.
- Mengambil dan menyimpan sebuah gambar sebagai *device-independent bitmap* (DIB).

Video *capture* mendigitalkan sebuah *stream data* audio-video dan menyimpannya ke dalam *hard disk* atau tempat penyimpanan lainnya. Akan dijelaskan bagaimana pengambilan gambar video pada sebuah aplikasi dengan menggunakan 3 pernyataan.

Kelas AVICap menangani secara detail pengambilan *streaming* audio dan video ke dalam file AVI. Hal ini akan membebaskan aplikasi dari kerumitan format file AVI, manajemen *buffer* video dan audio, dan akses *low-level* dari *driver* peralatan audio dan video. AVICap menyediakan sebuah antarmuka yang fleksibel untuk aplikasi yang dibuat. Baris-baris kode sederhana berikut merupakan contoh kode untuk mengambil gambar dari sebuah kamera:

```
hWndC = capCreateCaptureWindow( "My Capture Window",
        WS_CHILD | WS_VISIBLE,
        0, 0, 160, 120, hwndParent, nID);
```

Baris kode di atas digunakan untuk membuat sebuah *window* yang akan digunakan untuk menampilkan hasil pengambilan gambar dari kamera. Sedangkan untuk menghubungkan aplikasi dengan *driver* kamera serta memulai pengambilan gambar oleh kamera digunakan baris-baris kode berikut:

```
SendMessage(hWndC, WM_CAP_DRIVER_CONNECT, 0, 0L);
SendMessage(hWndC, WM_CAP_SEQUENCE, 0, 0L);
```

Selain menggunakan fungsi SendMessage untuk melakukan pengambilan gambar dari kamera, AVICap juga menyediakan beberapa antarmuka *macro* yang selain dapat digunakan untuk menggantikan fungsi SendMessage di atas, *macro* tersebut akan membuat baris kode lebih mudah dibaca.

```
capDriverConnect (hWndC, 0);
capCaptureSequence (hWndC);
```

Untuk merubah satu atau lebih parameter *capture* yang didefinisikan pada struktur CAPTUREPARMS, dapat dilakukan hal-hal sebagai berikut:

- Merubah c*apture rate*

  Capture rate adalah jumlah *frame* yang diambil tiap detik. Untuk memperoleh nilai *capture rate* digunakan *message* WM_CAP_GET_SEQUENCE_SETUP atau dengan menggunakan *macro* capCaptureGetSetup. Untuk menentukan nilai *capture rate* digunakan *message* WM_CAP_SET_SEQUENCE_SETUP atau dengan menggunakan *macro* capCaptureSetSetup. Nilai *capture rate* disimpan dalam *member* dwRequestMicroSecPerFrame. Nilai awal untuk *member* dwRequestMicroSecPerFrame adalah 66667, yang setara dengan 15 *frame* per detik.

- Menentukan kontrol untuk mengakhiri pengambilan gambar

  Pemakai diijinkan untuk melakukan pembatalan pengambilan gambar dengan menekan sebuah tombol pada *keyboard* atau mengklik tombol pada *mouse*. Jika pemakai melakukan pembatalan pada sebuah sesi pengambilan gambar *real-time*, maka isi dari file *capture* sampai titik pembatalan akan disimpan. Definisi *keystroke* disimpan dalam *member* vKeyAbort yang terdapat pada struktur CAPTUREPARMS, nilai awal *member* vKeyAbort adalah VK_ESCAPE. Sedangkan definisi tombol *mouse* disimpan dalam *member* fAbortLeftMouse dan fAbortRightMouse, dengan nilai awal pada keduanya adalah TRUE. Untuk memperoleh nilai dari *member-member* tersebut dapat digunakan *message* WM_CAP_GET_SEQUENCE_SETUP atau dengan menggunakan *macro* capCaptureGetSetup. Setelah melakukan perubahan pada *member-member* sesuai dengan keperluan, simpan perubahan yang terjadi dengan menggunakan *message* WM_CAP_SET_SEQUENCE_SETUP atau dengan menggunakan *macro* capCaptureSetSetup.

- Menentukan durasi untuk pengambilan gambar

  Kita dapat membatasi durasi dari operasi pengambilan gambar dengan menggunakan *member* fLimitEnabled dan wTimeLimit. Nilai awal untuk *member* fLimitEnabled adalah FALSE. Cara memperoleh dan menyimpan perubahan pada *member* fLimitEnabled dan wTimeLimit sama dengan yang dilakukan pada *member-member* struktur CAPTUREPARMS yang lain.

2.2.2. *Capture Driver*

Sebuah *capture driver* dan perangkat keras yang berhubungan dengan *driver* dapat menentukan beberapa aspek dari pengambilan gambar, termasuk kemampuan menerima video *source*, pilihan tampilan, format video, dan pilihan kompresi.

Untuk memperoleh kemampuan perangkat keras yang terhubung dengan *driver* dapat digunakan *message* WM_CAP_DRIVER_GET_CAPS atau dengan *macro* capDriverGetCaps. Kemampuan *capture driver* dan perangkat keras yang terhubung dengan *driver* tersebut disimpan dalam struktur CAPDRIVERCAPS.

2.2.3. Kotak Dialog Video

Tiap *capture driver* dapat menyediakan lebih dari empat kotak dialog untuk mengkontrol aspek video secara digital dan proses *capture*, dan untuk mendefinisikan atribut-atribut kompresi yang akan mengurangi ukuran dari data video.

Kotak dialog video *source* mengkontrol dalam pemilihan kanal-kanal masukan video dan parameter-parameter yang berpengaruh pada gambar video selama proses digitalisasi dalam *buffer*. Dalam kotak dialog ini menyediakan bermacam-macam tipe sinyal yang menghubungkan sumber video dengan *capture card* (biasanya SVHS), dan menyediakan kontrol-kontrol untuk mengubah hue, *contrast* dan saturasi. Jika kotak dialog ini didukung oleh *driver* video *capture*, maka untuk menampilkannya dapat dilakukan dengan menggunakan *message* WM_CAP_DLG_VIDEOSOURCE atau dengan *macro* capDlgVideoSource.

Kotak dialog video *format* mengkontrol dalam pemilihan dimensi *frame* dan kedalaman gambar video digital, dan pilihan penggunaan kompresi video.

Jika penggunaan kotak dialog ini didukung oleh *driver* video *capture*, maka untuk menampilkannya dapat digunakan *message* WM_CAP_DLG_VIDEOFORMAT atau dengan *macro* capDlgVideoFormat.

Kotak dialog video *display* mengkontrol tampilan video pada layar selama proses *capture*. Kontrol-kontrol pada kotak dialog ini tidak berpengaruh pada data video terdigitalisasi, tetapi berpengaruh pada penyajian sinyal terdigitalisasi. Jika kotak dialog ini didukung oleh *driver* video, maka untuk menampilkannya dapat digunakan *message* WM_CAP_DLG_VIDEODISPLAY atau dengan *macro* capDlgVideoDisplay.

Kotak dialog video *compression* mengkontrol atribut-atribut kompresi video. Jika kotak dialog ini didukung oleh *driver* video, maka untuk menampilkannya, digunakan *message* WM_CAP_DLG_VIDEOCOMPRESSION atau dengan *macro* capDlgVideoCompression.

2.2.4. Mode *Preview* dan *Overlay*

Sebuah *capture driver* dapat mengimplementasikan 2 metode untuk menampilkan sebuah video *stream*, mode *preview* dan mode *overlay*. Mode *preview* mentransfer *frame* digital dari perangkat keras *capture* ke memori sistem dan menampilkannya dalam sebuah *capture window* dengan menggunakan fungsi-fungsi g*raphics device interface* (GDI). Aplikasi mungkin akan mengurangi *preview rate* ketika *window* utama aplikasi kehilangan fokus, dan memperbesar *preview rate* ketika *window* utama aplikasi mendapat fokus. Semua yang terjadi pada mode ini, dipengaruhi oleh kinerja prosesor.

Dalam mode *preview*, terdapat 3 *message* untuk mengkontrol operasi *preview*:

- Gunakan *message* WM_CAP_SET_PREVIEW untuk menghidupkan atau mematikan mode *preview* dengan mengirimkan *message* tersebut ke *capture window*. Hal tersebut dapat dilakukan juga dengan memanggil *macro* capPreview. Menghidupkan mode ini, akan secara otomatis mematikan mode *overlay*.

- Gunakan *message* WM_CAP_SET_PREVIEWRATE (atau dapat juga dengan menggunakan *macro* capPreviewRate) untuk menentukan kecepatan dimana *frame* akan ditampilkan dalam mode *preview*.
- Gunakan *message* WM_CAP_SET_SCALE (dapat juga dengan menggunakan *macro* capPreviewScale) untuk menghidupkan atau mematikan penskalaan dari *preview* video. Ketika fasilitas ini dihidupkan, ukuran video *frame* yang ditampilkan akan disesuaikan dengan ukuran *capture window*.

Mode *overlay* akan menampilkan isi dari *capture buffer* ke layar tanpa menggunakan sumber daya CPU. Untuk menghidupkan atau mematikan mode ini dapat digunakan *message* WM_CAP_SET_OVERLAY atau dapat juga menggunakan *macro* capOverlay. Menghidupkan mode *overlay* akan secara otomatis mematikan mode *preview*.

2.2.5. Format Video dan Pengaturan Video *Capture*

Untuk dapat memperoleh struktur dari format video atau ukuran dari struktur tersebut dapat digunakan *message* WM_CAP_GET_VIDEOFORMAT atau menggunakan *macro* capGetVideoFormat dan capGetVideoFormatSize. Sedangkan untuk mengatur format video dapat dilakukan dengan menggunakan *message* WM_CAP_SET_VIDEOFORMAT atau dengan menggunakan *macro* capSetVideoFormat.

Struktur CAPTUREPARMS berisi parameter-parameter kontrol untuk *streaming* video *capture*. Struktur inilah yang mengkontrol beberapa aspek dari proses *capture* dan hal-hal berikut ini yang perlu diperhatikan pada waktu membuat aplikasi:

- Menentukan besarnya *frame rate*.
- Menentukan besar video *buffer* yang dialokasikan.
- Menghidupkan dan mematikan audio *capture*.
- Menentukan jeda waktu pada saat proses *capture*.
- Menentukan apakah MCI *device* (VCR atau videodisc) digunakan selama proses *capture*.
- Menentukan kontrol yang dipakai untuk mengakhiri proses *streaming*, dengan menggunakan *keyboard* atau *mouse*.

Untuk dapat memperoleh struktur CAPTUREPARMS dapat dilakukan dengan menggunakan *message* WM_CAP_GET_SEQUENCE_SETUP atau dengan *macro* capCaptureGetSetup. Dan tiap melakukan perubahan terhadap struktur tersebut, struktur harus disimpan kembali dengan memanggil *message* WM_CAP_SET_SEQUENCE_SETUP atau denga *macro* capCaptureSetSetup.

Untuk melakukan proses *capture* tanpa menuliskan datanya ke dalam *disk* dapat menggunakan *message* WM_CAP_SEQUENCE_NOFILE atau dengan *macro* capCaptureSequenceNoFile. *Message* ini baru bisa berfungsi penuh bila bekerja sama dengan fungsi-fungsi *callback*, yang memperbolehkan aplikasi menggunakan data video dan audio secara langsung.

Buffer yang digunakan video *capture* terletak dalam memori *heap*. Besar *buffer* yang digunakan dalam sebuah operasi *capture* dapat bervariasi dan tergantung nilai yang telah ditentukan pada *member* wNumVideoRequested dari struktur CAPTUREPARMS dan memori sistem yang tersedia.

2.2.6. Pengambilan *Buffer* Video

Pengambilan *buffer* video melibatkan beberapa fungsi dan *message*. Seperti telah dijelaskan pada bagian sebelumnya, bahwa untuk melakukan proses *capture* tanpa menuliskan data yang diperoleh oleh *capture driver* dari kamera ke dalam *disk* dapat menggunakan *message* WM_CAP_SEQUENCE_NOFILE atau dengan *macro* capCaptureSequenceNoFile.

2.2.6.1. *Macro* capCaptureSequenceNoFile

Macro capCaptureSequenceNoFile digunakan untuk memulai *streaming* video *capture* tanpa menuliskan data ke dalam *disk*. Sintaknya adalah sebagai berikut:

```
BOOL capCaptureSequenceNoFile(
      hwnd
      );
```

Parameter:

hwnd merupakan *handle* untuk sebuah *capture window*.

Macro ini akan mengembalikan nilai TRUE jika sukses dan FALSE jika sebaliknya. Pemanggilan terhadap *macro* ini belum bisa digunakan untuk

mengambil *buffer* video dari *capture driver*. Masih diperlukan lagi *macro* capSetCallbackOnVideoStream dan fungsi *callback* capVideoStreamCallback.

2.2.6.2. *Macro* capSetCallbackOnVideoStream

Macro capSetCallbackOnVideoStream berguna untuk menentukan fungsi *callback* mana yang dipakai aplikasi untuk memperoleh *buffer* video tersebut. *Macro* ini harus dijalankan terlebih dahulu sebelum proses *capture* dimulai. Sintak *macro* capSetCallbackOnVideoStream adalah sebagai berikut:

```
BOOL capSetCallbackOnVideoStream(
      hwnd,
      fpProc
      );
```

Parameter:

hwnd merupakan *handle* untuk sebuah *capture window*.

fpProc merupakan *pointer* yang menunjuk ke fungsi *callback* video-*stream*. Pemberian nilai NULL ke parameter ini berarti mematikan fungsi *callback* yang sudah dijalankan sebelumnya.

Jika *macro* ini berhasil melakukan tugasnya maka akan mengembalikan nilai TRUE, tapi jika sesi *streaming capture* masih dalam progres maka *macro* ini akan mengembalikan nilai FALSE.

2.2.6.3. Fungsi *Callback* capVideoStreamCallback

Fungsi *callback* capVideoStreamCallback digunakan pada waktu proses *streaming capture* untuk mengambil data tiap *frame*. Fungsi ini harus didaftarkan dulu dengan memanggil *macro* capSetCallbackOnVideoStream sebelum proses *capture* dimulai. Sintak fungsi *callback* ini sebagai berikut:

```
LRESULT CALLBACK capVideoStreamCallback(
      HWND hWnd,
      LPVIDEOHDR lpVHdr
      );
```

Parameter:

hWnd merupakan *handle* untuk sebuah *capture window* yang berhubungan dengan fungsi *callback*.

lpVHdr merupakan *pointer* yang menunjuk ke struktur VIDEOHDR. Struktur ini berisi informasi mengenai *frame* yang sedang diambil datanya.

AVICap memanggil fungsi *callback* capVideoStreamCallback ini selama proses *capture* berjalan ketika *buffer* video sudah terisi data yang baru. *Buffer* video dapat ditemukan pada struktur VIDEOHDR. Berikut ini penjelasan mengenai isi dari struktur VIDEOHDR tersebut:

```
typedef struct videohdr_tag {
    LPBYTE      lpData;
    DWORD       dwBufferLength;
    DWORD       dwBytesUsed;
    DWORD       dwTimeCaptured;
    DWORD       dwUser;
    DWORD       dwFlags;
    DWORD_PTR   dwReserved[4];
} VIDEOHDR, NEAR *PVIDEOHDR, FAR * LPVIDEOHDR;
```

lpData merupakan *pointer* yang menunjuk ke data *buffer*.

dwBufferLength menyatakan panjang dari data *buffer*.

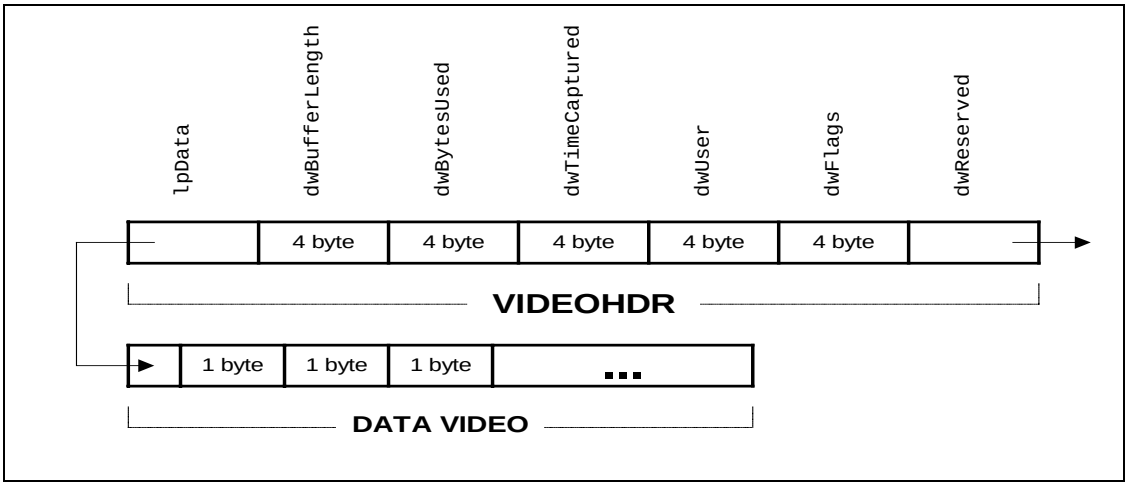
dwBytesUsed menyatakan jumlah *byte* yang dipakai.

dwTimeCaptured menyatakan lama waktu dalam *millisecond* (seper seribu detik) sejak proses *capture* dimulai.

dwUser didefinisikan sendiri untuk keperluan pemakai.

dwFlags nilai *member* ini dapat berupa nilai-nilai berikut: VHDR_DONE, VHDR_PREPARED, VHDR_INQUEUE dan VHDR_KEYFRAME.

dwReserved digunakan sebagai cadangan untuk *driver*.



Gambar 2.7. Struktur VIDEOHDR

2.3. DrawDib

Bagian ini membahas tentang DrawDib yang merupakan sekumpulan fungsi-fungsi yang akan digunakan oleh aplikasi *client* untuk menampilkan data video dalam bentuk gambar ke layar. Pembahasan dalam bagian ini hanya mencakup hal-hal yang ada pada DrawDib yang mendukung pembuatan aplikasi *client*.

2.3.1. Sekilas tentang DrawDib

Fungsi-fungsi DrawDib menyediakan kemampuan menampilkan gambar dengan performa tinggi untuk *device-independent bitmaps* (DIBs). Fungsi-fungsi DrawDib mendukung DIBs dengan kualitas gambar 8-bit, 16-bit, 24-bit dan 32-bit. Fungsi-fungsi DrawDib menulis ke secara memori video secara langsung sehingga tidak bergantung pada fungsi-fungsi *graphics device interface* (GDI).

Sekilas, fungsi-fungsi DrawDib mirip dengan fungsi StretchDIBits dalam hal kemampuan *image-stretching* dan *dithering*. Fungsi-fungsi DrawDib juga mendukung *image decompression* dan *data-streaming*.

Sebelum memulai akses pada fungsi-fungsi DrawDib, perlu dilakukan pemanggilan fungsi DrawDibOpen terlebih dahulu. Fungsi DrawDibOpen ini akan memuat sebuah *dynamic-link library* (DLL), mengalokasikan sejumlah *resource* memori, membuat sebuah DrawDib *device context* (DC) dan mempertahankan referensi ke sejumlah DC yang telah terinisialisasi. Fungsi DrawDibOpen juga mengembalikan sebuah *handle* dari DC baru yang akan digunakan pada pemanggilan fungsi-fungsi DrawDib yang lain.

Untuk menutup sebuah DrawDib DC setelah selesai menggunakannya, dapat memanggil fungsi DrawDibClose. Fungsi ini akan mengurangi sejumlah referensi dari pengaksesan DLL oleh aplikasi. Fungsi ini adalah fungsi DrawDib yang harus dipanggil paling akhir dalam aplikasi.

Setiap aplikasi dapat memiliki beberapa DrawDib DC, yang dapat menampilkan beberapa gambar bitmap secara bersamaan, dengan karakteristik yang berbeda. Dengan begitu, tiap DrawDib DC dapat diatur sesuai dengan kebutuhannya. Sebagai contoh, sebuah aplikasi yang mempunyai dua buah DrawDib DC, DC yang pertama digunakan untuk menampilkan sebuah gambar

dengan resolusi normal, sedangkan DC yang lain digunakan untuk menampilkan gambar dengan ukuran dan resolusi yang lebih besar.

Setelah sebuah DrawDib DC terbuat, untuk menggambar sebuah DIB ke layar dapat digunakan fungsi DrawDibDraw. Fungsi DrawDibDraw juga mampu menampilkan serangkaian *bitmaps* dengan dimensi dan format yang sama. Fungsi DrawDibBegin menyediakan DrawDibDraw sebuah DrawDib DC, sebuah *handle* DC, alamat yang menunjuk ke struktur BITMAPINFOHEADER serta dimensi sumber dan tujuan. Ketika serangkaian *bitmaps* ditampilkan, DrawDibDraw memeriksa nilai dari parameter-parameter di atas pada tiap-tiap gambar dalam serangkaian *bitmaps* tersebut. Jika DrawDibDraw mendapati perubahan nilai pada parameter-parameter tersebut, maka secara implisit fungsi DrawDibDraw memanggil kembali fungsi DrawDibBegin untuk menyesuaikan perubahan yang terjadi pada DrawDib DC.

2.3.2. Fungsi-fungsi dalam DrawDib

Tidak semua fungsi-fungsi dalam DrawDib akan dibahas dalam bagian ini, hanya fungsi-fungsi yang dipakai dalam aplikasi yang akan dijelaskan. Fungsi-fungsi tersebut antara lain: DrawDibOpen, DrawDibBegin, DrawDibDraw dan DrawDibClose. Berikut ini pembahasan secara lengkap tentang fungsi-fungsi tersebut:

2.3.2.1. Fungsi DrawDibOpen

Fungsi DrawDibOpen berfungsi untuk membuka *library* DrawDib. Pembukaan *library* DrawDib tersebut bertujuan supaya aplikasi bisa membuat dan menggunakan sebuah DrawDib DC untuk menampilkan gambar di layar.

```
HDRAWDIB DrawDibOpen(VOID);
```

Fungsi DrawDibOpen akan mengembalikan sebuah *handle* untuk DrawDib DC jika berhasil membuatnya, namun jika gagal maka akan mengembalikan nilai NULL.

2.3.2.2. Fungsi DrawDibBegin

Fungsi DrawDibBegin berfungsi menginisialisasi sebuah DrawDib DC baru serta melakukan perubahan nilai pada parameter-parameternya.

```
BOOL DrawDibBegin(
      HDRAWDIB hdd,
      HDC hdc,
      int dxDest,
      int dyDest,
      LPBITMAPINFOHEADER lpbi,
      int dxSrc,
      int dySrc,
      UINT wFlags
      );
```

Parameter:

hdd merupakan *handle* untuk DrawDib DC.

hdc merupakan *handle* untuk sebuah DC. Parameter ini merupakan pilihan.

dxDest merupakan lebar dari dimensi tujuan, dengan satuan MM_TEXT.

dyDest merupakan tinggi dari dimensi tujuan, dengan satuan MM_TEXT.

lpbi adalah *pointer* yang menunjuk ke struktur BITMAPINFOHEADER yang berisi format gambar yang akan ditampilkan. Tabel warna DIB mengikuti format gambar dan *member* biHeight harus bernilai positif.

dxSrc merupakan lebar dari dimensi sumber, dengan satuan *pixel*.

dySrc merupakan tinggi dari dimensi sumber, dengan satuan *pixel*.

wFlags dapat berisikan nilai-nilai yang telah didefinisikan berikut:

- DDF_ANIMATE: menyediakan animasi palet. Jika nilai ini disertakan, DrawDib mencadangkan tempat sebanyak mungkin dengan mengatur PC_RESERVED dalam *member* palPalEntry dari struktur LOGPALETTE dan palet ini dapat dianimasikan dengan menggunakan fungsi DrawDibChangePalette.
- DDF_BACKGROUNDPAL: palet digunakan untuk penggambaran sebagaimana permintaan *background*, membiarkan palet yang aktif digunakan untuk tampilan yang tidak berubah.
- DDF_BUFFER: menyebabkan DrawDib mencoba untuk menggunakan sebuah *off-screen buffer* sehingga DDF_UPDATE dapat digunakan. DDF_BUFFER menonaktifkan dekompresi dan penggambaran secara

langsung ke layar. Jika DrawDib tidak dapat membuat sebuah *off-screen buffer*, DrawDib akan melakukan dekompresi atau penggambaran secara langsung ke layar.

- DDF_DONTDRAW: gambar yang aktif sekarang tidak digambar, tetapi akan didekompres. DDF_UPDATE dapat digunakan kemudian untuk menampilkan sebuah gambar.
- DDF_JUSTDRAWIT: menampilkan sebuah gambar dengan menggunakan GDI. Mencegah fungsi-fungsi DrawDib dari dekompresi, *stretching* ataupun *dithering* gambar. DDF_JUSTDRAWIT inilah yang membedakan kemampuan DrawDib dari fungsi StretchDIBits.
- DDF_SAME_DRAW: menggunakan parameter-parameter penggambaran yang aktif untuk DrawDibDraw. Penggunaan DDF_SAME_DRAW hanya jika parameter lpbi, dxDest, dyDest, dxSrc dan dySrc tidak berubah sejak fungsi DrawDibBegin dan DrawDibDraw digunakan.
- DDF_SAME_HDC: menggunakan *handle* DC yang aktif dan palet secara aktif diasosiasikan dengan DC.
- DDF_UPDATE: *buffer bitmap* terakhir perlu digambar ulang. Jika penggambaran gagal maka sebuah *buffer* gambar tidak tersedia dan sebuah gambar baru perlu disebutkan sebelum tampilan dapat diperbarui.

Fungsi ini akan mengembalikan nilai TRUE jika berhasil dan jika sebaliknya akan mengembalikan nilai FALSE. Fungsi ini melakukan persiapan untuk menggambar sebuah DIB yang disebutkan pada parameter lpbi ke DC. Gambar akan direntangkan sesuai dengan ukuran yang disebutkan pada parameter dxDest dan dyDest. Jika dxDest dan dyDest diberi nilai –1 maka DIB akan digambar pada skala 1:1 tanpa direntangkan.

2.3.2.3. Fungsi DrawDibDraw

Fungsi DrawDibDraw berfungsi menggambar sebuah DIB ke layar.

```
BOOL DrawDibDraw(
      HDRAWDIB hdd,
      HDC hdc,
      int xDst,
      int yDst,
      int dxDst,
```

```
int dyDst,
LPBITMAPINFOHEADER lpbi,
LPVOID lpBits,
int xSrc,
int ySrc,
int dxSrc,
int dySrc,
UINT wFlags
);
```

Parameter:

hdd merupakan *handle* untuk DrawDib DC.

hdc merupakan *handle* untuk sebuah DC.

xDst merupakan koordinat x, yang menentukan koordinat x pojok kiri atas jendela tujuan.

yDst merupakan koordinat y, yang menentukan koordinat y pojok kiri atas jendela tujuan.

dxDst merupakan lebar dari jendela tujuan. Jika dxDst bernilai –1 maka lebar dari *bitmap* yang akan dipakai.

dyDst merupakan tinggi dari jendela tujuan. Jika dyDst bernilai –1 maka tinggi dari *bitmap* yang akan dipakai.

lpbi merupakan *pointer* yang menunjuk ke struktur BITMAPINFOHEADER yang berisi format gambar.

lpBits merupakan *pointer* yang menunjuk ke *buffer* yang berisi bit-bit dari *bitmap*.

xSrc merupakan koordinat x, yang menentukan koordinat x pojok kiri atas jendela sumber dalam satuan *pixel*.

ySrc merupakan koordinat y, yang menentukan koordinat y pojok kiri atas jendela sumber dalam satuan *pixel*.

dxSrc merupakan lebar dari jendela sumber dalam satuan *pixel*.

dySrc merupakan tinggi dari jendela sumber dalam satuan *pixel*.

wFlags lihat parameter wFlags pada fungsi DrawDibBegin.

Fungsi DrawDibDraw akan mengembalikan nilai TRUE jika operasinya berhasil dan FALSE jika gagal. DDF_DONTDRAW akan menyebabkan fungsi DrawDibDraw melakukan dekompresi sebuah gambar tetapi tidak menampilkan gambar tersebut. Jika DrawDib DC tidak memiliki sebuah *off-screen buffer*,

menyebutkan DDF_DONTDRAW menyebabkan sebuah *frame* akan digambar ke layar sesegera mungkin.

2.3.2.4. Fungsi DrawDibClose

Fungsi DrawDibClose berfungsi menutup penggunaan sebuah DrawDib DC dan membebaskan semua *resource* DrawDib yang telah dialokasikan.

```
BOOL DrawDibClose(
      HDRAWDIB hdd
      );
```

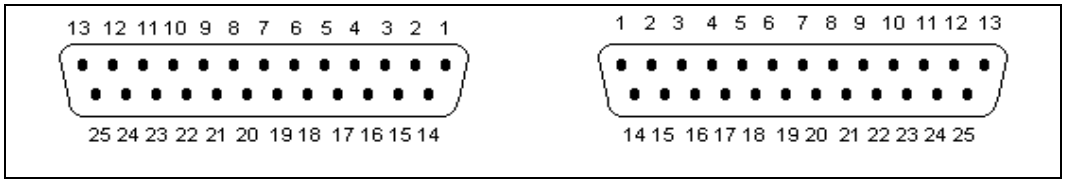
Parameter:

hdd merupakan *handle* untuk DrawDib DC.

Jika penutupan DrawDib berhasil maka fungsi DrawDibClose akan mengembalikan nilai TRUE, dan jika sebaliknya akan mengembalikan nilai FALSE.

2.4. Paralel *Port*

Paralel *port* merupakan standar koneksi antara sebuah *device* (biasanya *printer*) ke sebuah komputer. *Port* ini merupakan sebuah *port* I/O (Input/Output). Jadi melalui *port* ini, komputer tidak hanya bisa mengirim data ke *device*, melainkan juga bisa menerima data dari *device* yang terhubung dengannya. Gambar 2.8 adalah gambar penampang konektor paralel *port* untuk konektor *male* dan konektor *female*.



Gambar 2.8. Penampang Konektor Paralel *port*

Paralel *port* mempunyai 25 pin, yang secara garis besar terbagi menjadi tiga bagian, yaitu: data, kontrol dan status. Bagian data terdiri dari 8 pin (8-bit) dengan alamat *port* 378h, bagian status yang terdiri dari 5 pin (5-bit) dengan alamat *port* 379h dan bagian kontrol terdiri dari 4 pin (4-bit). Sedangkan 8 pin

yang tersisa merupakan *ground*. Tabel 2.1 menjelaskan konfigurasi pin-pin pada paralel *port*.

Tabel 2.1. Konfigurasi Pin-Pin Paralel *Port*

PIN	SIGN	TYPE	USED	SENSE
1	Control 0	Output	NSTROBE	Inverted
2	Data 0	Output	Data 0	Direct
3	Data 1	Output	Data 1	Direct
4	Data 2	Output	Data 2	Direct
5	Data 3	Output	Data 3	Direct
6	Data 4	Output	Data 4	Direct
7	Data 5	Output	Data 5	Direct
8	Data 6	Output	Data 6	Direct
9	Data 7	Output	Data 7	Direct
10	Status 6	Input	NACK	Direct
11	Status 7	Input	Busy	Inverted
12	Status 5	Input	Paper Empty	Direct
13	Status 4	Input	Select	Direct
14	Control 1	Output	nAUTOFEED	Inverted
15	Status 3	Input	NERROR	Direct
16	Control 2	Output	InitializePrinter	Direct
17	Control 3	Output	nSelect Input	Inverted
18-25	Ground			

Tabel 2.2. Konfigurasi *Port* 378h

PIN	SIGN	2^n = decimal
2	Data 0	$2^1 = 1$
3	Data 1	$2^2 = 2$
4	Data 2	$2^3 = 4$
5	Data 3	$2^4 = 8$
6	Data 4	$2^5 = 16$
7	Data 5	$2^6 = 32$
8	Data 6	$2^7 = 64$
9	Data 7	$2^8 = 128$

Tabel 2.2 menunjukkan urutan bobot tiap-tiap bit pada *port* 378h. *Port* 378h terdiri dari 8-bit, yang dimulai dari pin ke-2 (Data 0) dengan bobot bit terkecil hingga pin ke-9 (Data 7) dengan bobot bit terbesar.

Tabel 2.3. Konfigurasi *Port* 379h

PIN	SIGN	2^n = decimal
X	X	$2^1 = 1$
X	X	$2^2 = 2$
X	X	$2^3 = 4$
15	NERROR	$2^4 = 8$
13	Select	$2^5 = 16$
12	Paper Empty	$2^6 = 32$
10	NACK	$2^7 = 64$
11	Busy	$2^8 = 128$

Tabel 2.3 menjelaskan konfigurasi pin-pin pada *port* 379h beserta bobot tiap-tiap bitnya. Secara fisik, *port* 379h hanya memiliki 5-bit, namun secara perhitungan, *port* ini terdiri dari 8-bit. Tiga bobot bit terendah, yaitu dengan nilai desimal 1, 2 dan 4 merupakan bit-bit yang secara fisik tidak ada. Bit-bit ini dapat diberi logika apapun, tergantung keinginan pembuat sistem. Dengan demikian, perlu berhati-hati dalam perhitungan bobot bit-bit pada *port* 379h ini, sebab jika keliru dalam perhitungannya maka logika untuk program yang dibuat juga akan keliru.

Tabel 2.4. Konfigurasi *Port* 37Ah

PIN	SIGN	2^n = decimal
1	NSTROBE	$2^1 = 1$
14	NAUTOFEED	$2^2 = 2$
16	Initialize	$2^3 = 4$
17	NSELECT_IN	$2^4 = 8$
X	X	$2^5 = 16$
X	X	$2^6 = 32$
X	X	$2^7 = 64$
X	X	$2^8 = 128$

Sama halnya dengan *port* 379h, secara fisik, *port* 37Ah hanya memiliki 4-bit, namun secara perhitungan, *port* ini terdiri dari 8-bit. 4-bit yang dimiliki *port* ini menempati pin ke-1, ke-14, ke-16 dan ke-17. Konfigurasi pin-pinnya secara lengkap dapat dilihat ada tabel 2.4.