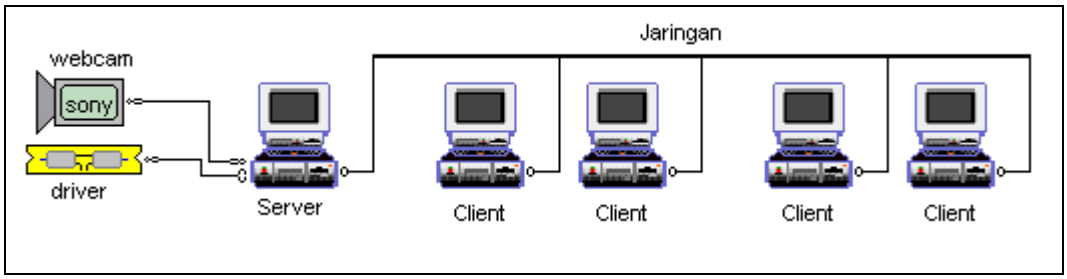
3. PERANCANGAN DAN IMPLEMENTASI SISTEM

Sistem keamanan yang akan dibahas pada bagian berikut ini merupakan sistem keamanan yang mengandalkan teknologi informasi, karena dalam sistem tersebut membutuhkan teknologi *internet* ataupun *intranet* sebagai media yang menghubungkan dua tempat, yaitu tempat yang akan dipantau sebagai *server* dan tempat di mana pemantau berada sebagai *client*. Sistem yang dirancang ini juga memanfaatkan *webcam* dan alat penggerak berupa rangkaian *driver* dan motor DC yang dihubungkan dengan komputer *server*.



Gambar 3.1. Skema Umum Sistem

Gambar di atas menunjukkan skema umum sistem keamanan dengan menggunakan kamera yang akan dibuat. Kamera untuk keamanan berada pada sisi *server* yang dapat dikendalikan dan dipantau oleh sisi *client* melalui jaringan. Kamera yang dipakai pada sistem keamanan ini adalah jenis *webcam* yang terhubung dengan komputer *server* melalui USB *port* sedangkan *driver* penggerak terhubung dengan komputer *server* melalui paralel *port*.

Komputer *server* yang terhubung dengan *webcam* bertugas mengambil data video *streaming* dari *webcam* dan mengirimkannya ke komputer *client* melalui kabel jaringan. Komputer *client* yang terhubung dengan komputer *server* dapat lebih dari satu unit.

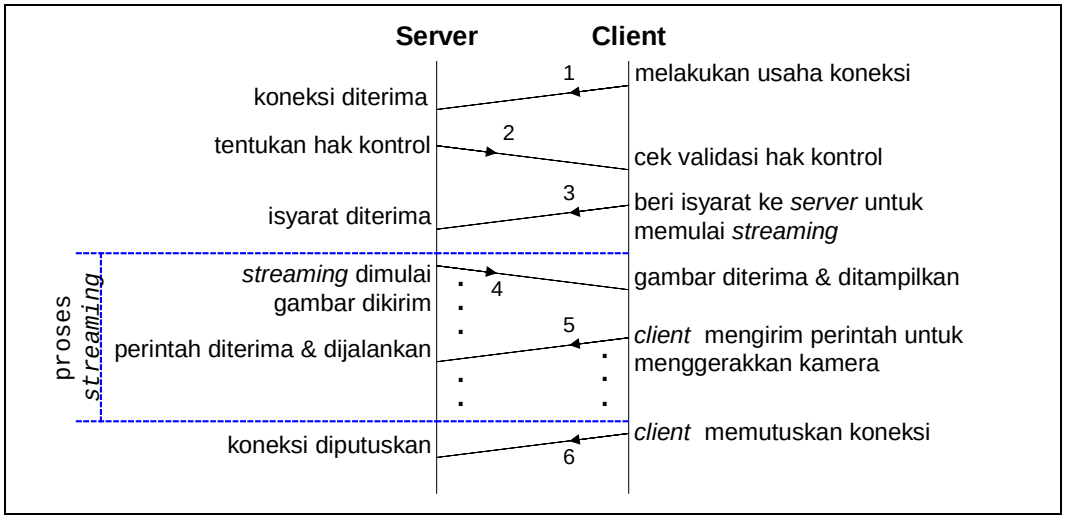
Secara garis besar, perancangan sistem keamanan dengan menggunakan kamera ini dapat dibagi menjadi dua bagian, yaitu perancangan *software* atau

perangkat lunak yang terdiri dari aplikasi *server* dan aplikasi *client* serta perancangan *hardware* atau perangkat keras yaitu rangkaian *driver* motor.

3.1. Perancangan *Software*

Perancangan *software* terdiri dari dua bagian, yaitu aplikasi *server* dan aplikasi *client*. Aplikasi *server* akan digunakan oleh komputer *server* untuk mengambil data *streaming* video dari *webcam*, melakukan pengikatan *socket* dan menunggu koneksi dari komputer *client*, mengirimkan data video *streaming* yang diperoleh dari *webcam* dan menggerakkan motor ketika ada perintah dari komputer *client*. Aplikasi *client* akan digunakan oleh komputer *client* untuk melakukan koneksi ke komputer *server*, menerima data video *streaming* dari *server* dan menampilkannya di layar serta mengirimkan perintah ke *server* untuk menggerakkan kamera.

Sebelum melakukan pembuatan aplikasi, baik aplikasi *server* maupun aplikasi *client*, sebaiknya ditentukan dulu protokol yang akan mengatur hubungan antara dua aplikasi tersebut. Adapun gambaran protokol koneksi antara komputer *client* dan *server* akan dijelaskan pada gambar 3.2.



Gambar 3.2. Protokol Koneksi antara *Client* dan *Server*

Setelah *server* dijalankan dan siap menerima koneksi, *client* berusaha menjalin koneksi dengan *server*, pada gambar 3.2 ditunjukkan dengan panah nomor 1. Pada saat koneksi telah terbentuk, *server* memeriksa apakah pada saat

itu ada *client* lain yang memiliki hak kontrol. Bila sudah ada *client* lain yang memiliki hak kontrol maka *client* yang baru terhubung dengan *server* tidak berhak menerima hak kontrol tersebut. Hanya *client* yang memiliki hak kontrol yang dapat memberikan perintah kepada *server* untuk menggerakkan kamera dan pada suatu saat hanya satu *client* saja yang memiliki hak kontrol tersebut sedangkan *client* yang lain hanya dapat menikmati hasil *capture* kamera.

Panah nomor 2 menyatakan pesan *server* kepada *client* yang akan memberitahukan berhak atau tidaknya *client* menerima hak kontrol. Setelah *Client* menerima pesan tersebut, apliaksi *client* memeriksa apakah hak kontrol diberikan kepada *client* atau tidak. Pada aplikasi *client* yang memiliki hak kontrol, tombol-tombol arah (kanan, kiri, atas dan bawah) dihidupkan, sedangkan pada aplikasi *client* yang tidak memiliki hak kontrol, tombol-tombol arah tersebut dimatikan.

Memiliki atau tidak hak kontrol, *client* harus tetap melanjutkan langkah berikutnya, yaitu memberitahukan kepada *server* untuk segera memulai proses *streaming*. Pemberitahuan ini ditunjukkan dengan panah nomor 3. Setelah aplikasi *server* menerima pemberitahuan ini, maka sesegera mungkin proses *streaming* dijalankan oleh *server* dengan mengirimkan data video ke *client*. Data video akan terus dikirimkan ke *client* selama koneksi antara *client* dan *server* belum terputus. Panah nomor 4 menunjukkan pengiriman data video dari *server* ke *client*. Pada sisi *client*, data video yang diterima dari *server* ditampilkan pada layar dalam bentuk gambar bergerak sehingga *user* pada sisi *client* bisa memantau kejadian apa saja yang gambarnya ditangkap oleh kamera *server*.

Selain dapat menerima gambar yang berupa data video dari *server* dan menampilkannya di layar, aplikasi *client* juga dapat memberikan perintah kepada *server* untuk menggerakkan motor yang terpasang pada kamera *server*. Namun, seperti yang telah dijelaskan pada bagian sebelumnya, bahwa hanya *client* tertentu saja yang dapat memberikan perintah tersebut, yaitu *client* yang memiliki hak kontrol. Perintah yang diberikan komputer *client* ke komputer *server* ditunjukkan oleh panah nomor 5.

Ketika perintah dari *client* untuk menggerakkan kamera diterima *server*, dengan sesegera mungkin *server* mengeluarkan sinyal ke paralel *port* yang isinya berupa bit-bit yang mengatur arah pergerakan motor serta fungsi yang mengatur

mati dan hidupnya motor. Pada kamera *server* terdapat 2 buah motor, motor yang mengatur gerakan kamera secara horisontal (motor horisontal) dan motor yang mengatur gerakan kamera secara vertikal (motor vertikal). Motor horisontal menggerakkan kamera ke arah kanan dan kiri, sedangkan motor vertikal menggerakkan kamera ke arah atas dan bawah.

Panah nomor 6 menunjukkan isyarat yang digunakan oleh *client* untuk memberitahukan *server* bahwa koneksi yang telah menghubungkan keduanya akan segera diputuskan. Aplikasi *client* mengirimkan suatu pesan yang berisi pemberitahuan pemutusan koneksi tersebut ke aplikasi *server*. Aplikasi *server* menerima pesan *client* tersebut dan memutuskan koneksi dengan *client* yang bersangkutan. Jika ternyata *client* yang memutuskan koneksi tersebut memiliki hak kontrol maka hak kontrol tersebut akan diberikan kepada *client* lain yang terhubung dengan *server* setelah pemutusan koneksi tersebut.

3.1.1. Perancangan dan Implementasi Aplikasi *Server*

Aplikasi *server* dirancang untuk melakukan hal-hal sebagai berikut:

- Melakukan koneksi dengan *webcam* melalui *driver webcam* dan mengaktifkan *capture* pada *webcam*.
- Membuat *socket* dan melakukan proses *listen* terhadap koneksi *client*.
- Menerima koneksi yang masuk dari *client* dan menentukan pemberian hak kontrol pada *client*.
- Mengambil data video pada *buffer memory* dan mengirimkannya ke komputer *client* secara *streaming*.
- Menerima perintah dari komputer *client* yang memiliki hak kontrol untuk menggerakkan kamera.
- Memutuskan koneksi dengan *client*.

Pemrograman *socket* dilakukan dengan menggunakan winsock dalam bahasa C++ dengan kompiler yang dipakai adalah Microsoft Visual C++ versi 6. Aplikasi ditulis dengan bantuan fasilitas *library* Microsoft Foundation Class (MFC), yang mempermudah dalam pembuatan *user interface* untuk aplikasi. Jenis protokol yang dipakai adalah TCP karena jenis koneksi yang digunakan adalah *connection-oriented*.

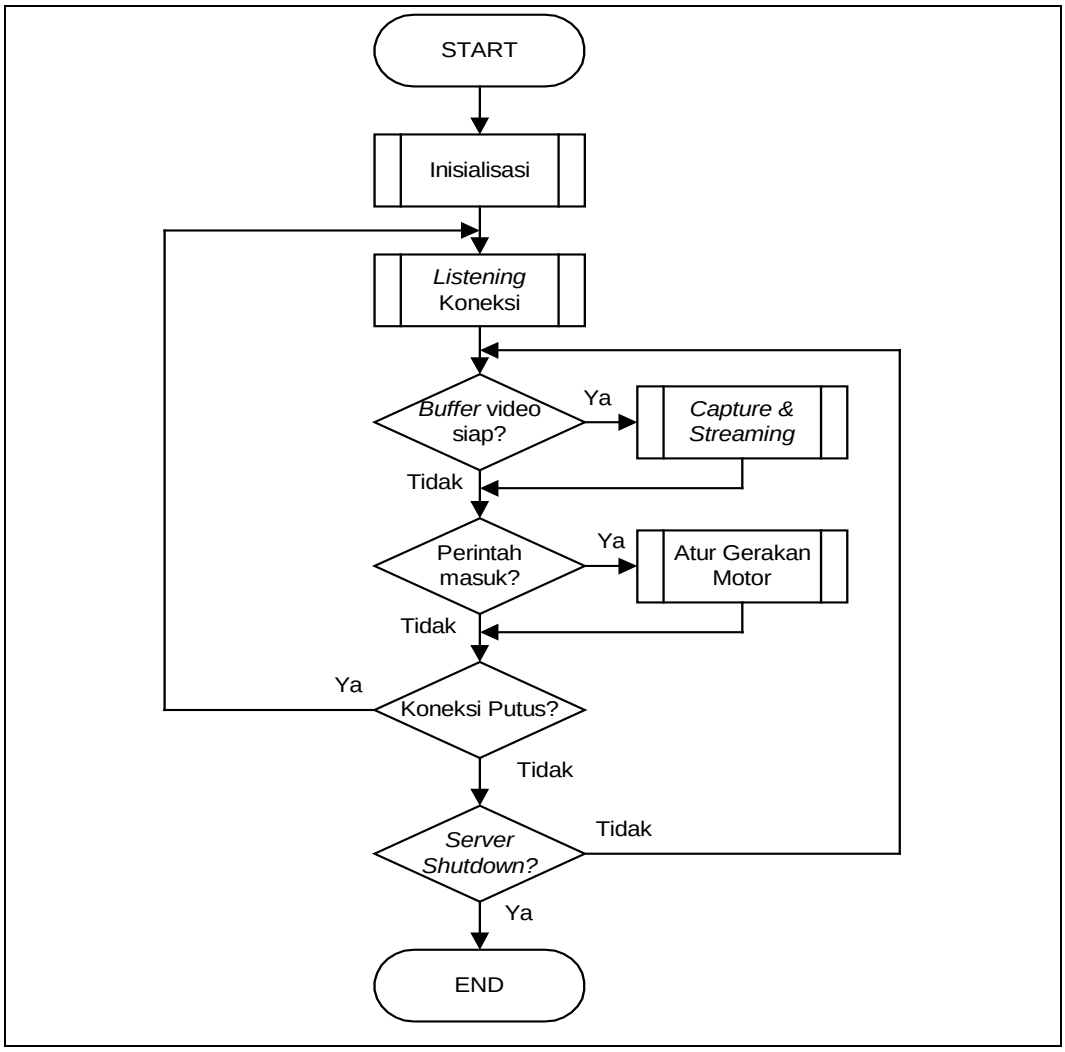
Dalam hal koneksi dengan *driver webcam*, aplikasi menggunakan sebuah *library* yang didalamnya telah membungkus fungsi-fungsi *capture* milik Windows dalam sebuah kelas yaitu kelas AviCap. *Library* tersebut terdapat dalam file cavicap.h dan implementasinya ada pada file cavicap.cpp. Dengan adanya kelas AviCap, sangat membantu sekali dalam melakukan koneksi dengan *driver webcam* dan sekaligus pada waktu mengaksesnya.

Karena aplikasi *server* yang dirancang lebih diperuntukan dan disarankan digunakan pada Windows NT, Windows 2000 dan Windows XP maka aplikasi juga akan dilengkapi dengan beberapa fungsi yang akan mendukung pengaksesan secara langsung ke paralel *port*. Tanpa fungsi-fungsi tersebut, aplikasi tidak bisa mengakses paralel *port* bila dijalankan pada berbagai sistem operasi di atas.

Gambar 3.3. *Screen Shot* Aplikasi *Server*

Gambar 3.3 merupakan bentuk tampilan aplikasi *server*. Pada sisi kiri tampilan aplikasi tersebut, terdapat sebuah kontrol *text field* yang digunakan untuk memasukkan nomor *port* yang diinginkan. Di bawahnya terdapat 2 buah kontrol *button*, kontrol *button* yang atas digunakan untuk mengawali dan mengakhiri pekerjaan *server*. Di sebelh kanan terdapat jendela *preview*, dalam jendela tersebut ditampilkan gambar yang sedang ditangkap kamera.

Setelah mengetahui rancangan aplikasi, ada baiknya bila pembahasan dialihkan ke sistem kerjanya. Sistem kerja aplikasi *server* akan dijelaskan dalam diagram alur berikut ini:



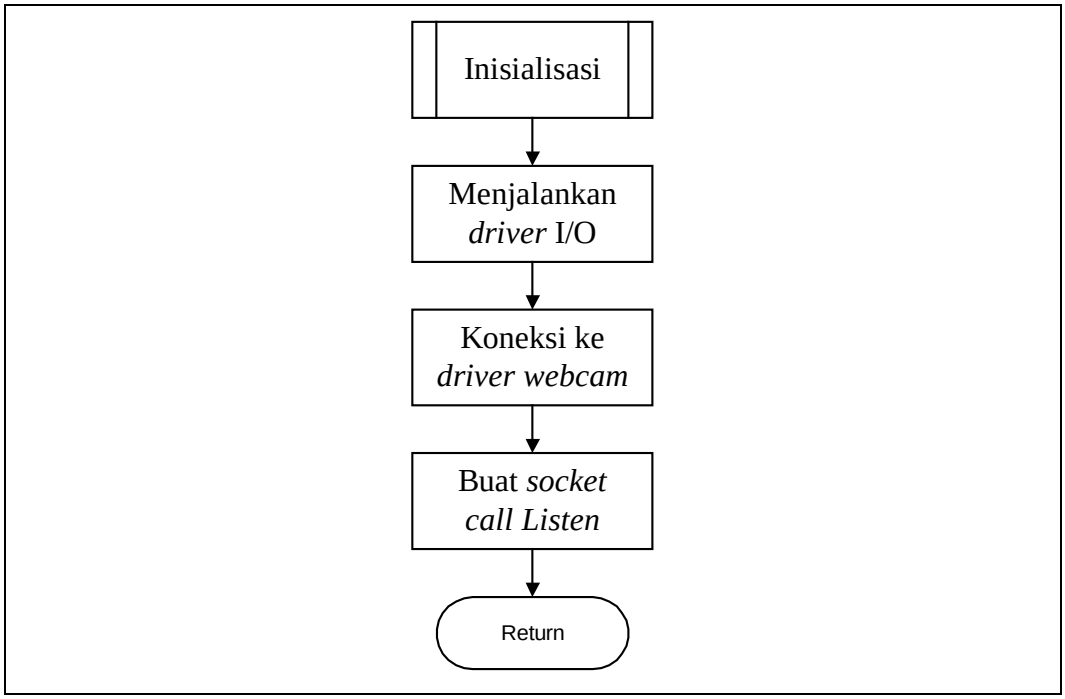
Gambar 3.4. Diagram Alur Aplikasi *Server*

Diagram alur aplikasi *server* di atas menggambarkan alur kerja dari aplikasi *server*. Di dalam diagram alur tersebut terdapat beberapa prosedur, seperti inisialisasi, *listening* koneksi, *capture* & *streaming* dan atur gerakan motor. Di dalam tiap-tiap prosedur tersebut terdapat proses-proses, yang menggambarkan apa yang sebenarnya dikerjakan oleh aplikasi. Berikut ini penjelasan proses-proses yang terjadi dalam setiap prosedur:

3.1.1.1. Prosedur Inisialisasi

Prosedur inisialisasi ini melakukan hal-hal yang berhubungan dengan pembuatan *socket server*, koneksi ke *driver webcam* serta mempersiapan Windows NT/XP agar bisa mengakses langsung ke I/O *port*.

Diagram alur untuk prosedur ini adalah sebagai berikut:



Gambar 3.5. Diagram Alur Prosedur Inisialisasi Aplikasi *Server*

Driver I/O *port* diperlukan bila komputer yang kita pakai sebagai *server* menggunakan sistem operasi Windows NT/2000/XP. Windows NT/2000/XP tidak mendukung pengalamatan secara langsung paralel *port*. *Driver* ini berfungsi untuk mendukung pengalamatan secara langsung paralel *port* pada Windows NT/2000/XP. *Driver* ini tidak akan berpengaruh bila komputer *server* yang dipakai menggunakan sistem operasi Windows 98.

Supaya *driver* I/O *port* tersebut bisa berjalan dengan baik maka salin file UserPort.sys ke direktori %windir%\system32\drivers sebelum menjalankan kode program berikut:

```
Initial_DriverNT();
ReadRegistry_DriverNT();
StartDriver_DriverNT();
```

Fungsi `Initial_DriverNT()` menuliskan alamat I/O *port* yang digunakan oleh aplikasi yaitu alamat 378h dan 379h pada *registry*. Hal ini dilakukan supaya ketika fungsi `ReadRegistry_DriverNT()` dijalankan, data pada HKEY_LOCAL_MACHINE\Software\UserPort sudah terisi alamat I/O *port* yang dibutuhkan. Setelah alamat I/O *port* dimuat, maka untuk menjalankan *driver*, aplikasi harus memanggil fungsi `StartDriver_DriverNT()`. Kode lengkap tiga fungsi di atas dapat dilihat pada lampiran.

Setelah *driver* I/O dijalankan, maka langkah berikutnya adalah membuat koneksi ke *driver webcam*. Aplikasi melakukan koneksi ke *driver webcam* dengan memakai variabel capwnd yang bertipe AviCap. Variabel tersebut dideklarasikan pada kelas CServerDlg dalam file ServerDlg.h. AviCap merupakan sebuah kelas yang menangani koneksi dan akses ke *webcam*. Koneksi ke *driver webcam* dilakukan dengan *method* ConnectWithDriver yang merupakan *method* milik kelas AviCap.

Selain membuat koneksi ke *driver webcam*, perlu juga dilakukan beberapa inisialisasi, seperti mengatur besarnya resolusi dan ukuran *frame*. Besar resolusi *frame* ditentukan dengan menggunakan *method* SetBitResolution miliki kelas AviCap. Sedangkan untuk menentukan ukuran dari *frane* dapat dilakukan dengan memanggil *method* SetFrameSize yang juga milik kelas AviCap. Berikut potongan kode untuk inisialisasi *driver webcam*:

```
if (capwnd.ConnectWithDriver(ANY_DRIVER))
{
      capwnd.SetFrameSize(160,120);
      capwnd.SetBitResolution(24);
      . . .
}
```

Method ConnectWithDriver melakukan pemanggilan terhadap fungsi capDriverConnect, yang merupakan fungsi milik Windows API. Parameter ANY_DRIVER yang disertakan pada *method* tersebut dimaksudkan agar semua jenis *driver* dikenali. Sedangkan *method* SetFrameSize melakukan pengubahan nilai pada *member* biWidth dan *member* biHeight yang merupakan milik struktur BITMAPINFOHEADER. Dalam hal ini, *method* SetFrameSize mengubah ukuran *frame* menjadi 160 x 120.

Terakhir, ditentukan pula besar resolusi *frame* dengan menggunakan *method* SetBitResolution. *Method* ini melakukan perubahan terhadap nilai dari *member* biBitCount milik struktur BITMAPINFOHEADER. Untuk keperluan aplikasi, maka besar resolusi yang dikehendaki adalah 24-bit, artinya besar data untuk tiap *pixel* adalah 24-bit atau 3 *byte*, 8-bit (1 *byte*) untuk porsi warna merah (R), 8-bit (1 *byte*) untuk porsi warna hijau (G) dan 8-bit (1 *byte*) untuk porsi warna biru (B). Dengan meninjau ukuran dan resolusi *frame*, maka besar *buffer* tiap *frame* yang diperoleh dari hasil *capture* sebesar 160 x 120 x 3 *byte* = 57600 *byte*.

Bagian berikut akan menjelaskan pembuatan *socket* pada aplikasi *server* dengan menggunakan winsock 2. Ada dua macam *socket* yang akan dibuat pada aplikasi *server*, yaitu *socket* untuk melakukan proses *listen* dan *socket* yang akan digunakan untuk melakukan koneksi dengan *client*. *Socket* untuk proses *listen* diberi nama sListen yang hanya berjumlah 1, sedangkan *socket* untuk koneksi dengan *client* diberi nama sockClient yang merupakan sebuah *array* sehingga *socket* ini jumlahnya dapat lebih dari 1.

Sebuah *socket* membutuhkan sebuah struktur sockaddr_in untuk menentukan alamat *endpoint* baik lokal maupun *remote* dimana *socket* akan terhubung. Karena ada 2 macam *socket* yang dibuat, maka terdapat juga 2 buah struktur sockaddr_in yaitu, sockaddr_in dengan nama saListen untuk *socket* sListen dan sockaddr_in dengan nama saClient untuk *socket* sockClient. Deklarasi 2 buah *socket* beserta 2 buah struktur sockaddr_in tersebut berada pada deklarasi kelas CServerDlg pada file ServerDlg.h. Berikut potongan kode programnya:

```
class CServerDlg : public Cdialog
{
public:
      . . .
      SOCKET sockClient[10];
      SOCKET sListen;
      struct sockaddr_in saListen;
      struct sockaddr_in saClient;
      . . .
}
```

Setelah mendeklarasikan semua *socket* dan struktur sockaddr_in-nya, akan dibahas bagaimana pembuatan *socket* dalam Microsoft Visual C++. Rincian proses pembuatan *socket* terdapat pada fungsi startServer yang merupakan milik kelas CServerDlg. Perhatikan potongan kode berikut:

```
sListen = socket(AF_INET, SOCK_STREAM, 0);
if ( sListen == INVALID_SOCKET )
{
      MessageBox("Failed to create socket!", "Server",
      MB_OK+MB_ICONERROR);
      return FALSE;
}
```

Dalam potongan kode program di atas, *socket* sListen diciptakan dengan menggunakan fungsi *socket*. Parameter AF_INET pada fungsi tersebut merupakan alamat *family*. Parameter SOCK_STREAM digunakan karena jenis protokol yang dipakai adalah TCP. Jika *socket* gagal diciptakan (*socket* masih dalam keadaan INVALID_SOCKET) maka akan ditampilkan pesan “Failed to create socket!”.

Bagian ini akan menjelaskan pengisian nilai pada struktur saListen serta bagaimana mengikatkan *socket* sListen pada sebuah alamat yang telah ditentukan pada struktur saListen. Nomor *port* dari *server* yang ada pada *member variable* m_port dimasukkan sebagai nilai dari *member* sin_port. Pengisian nilai pada *member* struktur saListen berguna pada langkah selanjutnya, yaitu pada pemanggilan fungsi *bind*.

Pemanggilan fungsi *bind* ini akan menyebabkan *socket* sListen diikatkan pada alamat yang ditentukan oleh struktur saListen dan dikenali oleh *client* melalui nomor *port*-nya. Jika *socket* gagal diikatkan dengan fungsi *bind* maka aplikasi akan menampilkan pesan “Failed to bind socket!”. Berikut ini potongan kode programnya:

```
saListen.sin_addr.s_addr = htonl(INADDR_ANY);
saListen.sin_family = AF_INET;
saListen.sin_port = htons(atoi(m_port));

if ( bind(sListen, (struct sockaddr*)&saListen, sizeof(saListen))
== SOCKET_ERROR )
{
      MessageBox("Failed to bind socket!", "Server",
      MB_OK+MB_ICONERROR);
      return FALSE;
}
```

Berikut ini akan dijelaskan bagaimana mendeklarasikan sebuah fungsi yang digunakan untuk merespon *message-message* yang berhubungan dengan jaringan seperti FD_ACCEPT yang digunakan untuk merespon bila ada *client* yang berhasil melakukan koneksi dengan *server* dan FD_READ yang digunakan untuk merespon bila ada data yang masuk ke *server*. Fungsi WSAAsyncSelect

berhubungan erat dengan pekerjaan ini. Nilai WM_USER_ASYNC_SELECT pada parameter ke-3 fungsi WSAAsyncSelect tidak lain merupakan nama *message* untuk menerima *event-event* yang terjadi pada jaringan. Berikut ini potongan kode programnya:

```
if ( WSAAsyncSelect(sListen, m_hWnd, WM_USER_ASYNC_SELECT,
FD_ACCEPT | FD_READ ) == SOCKET_ERROR )
{
      MessageBox("Server could not do async select!", "Server",
      MB_OK+MB_ICONERROR);
      return FALSE;
}
```

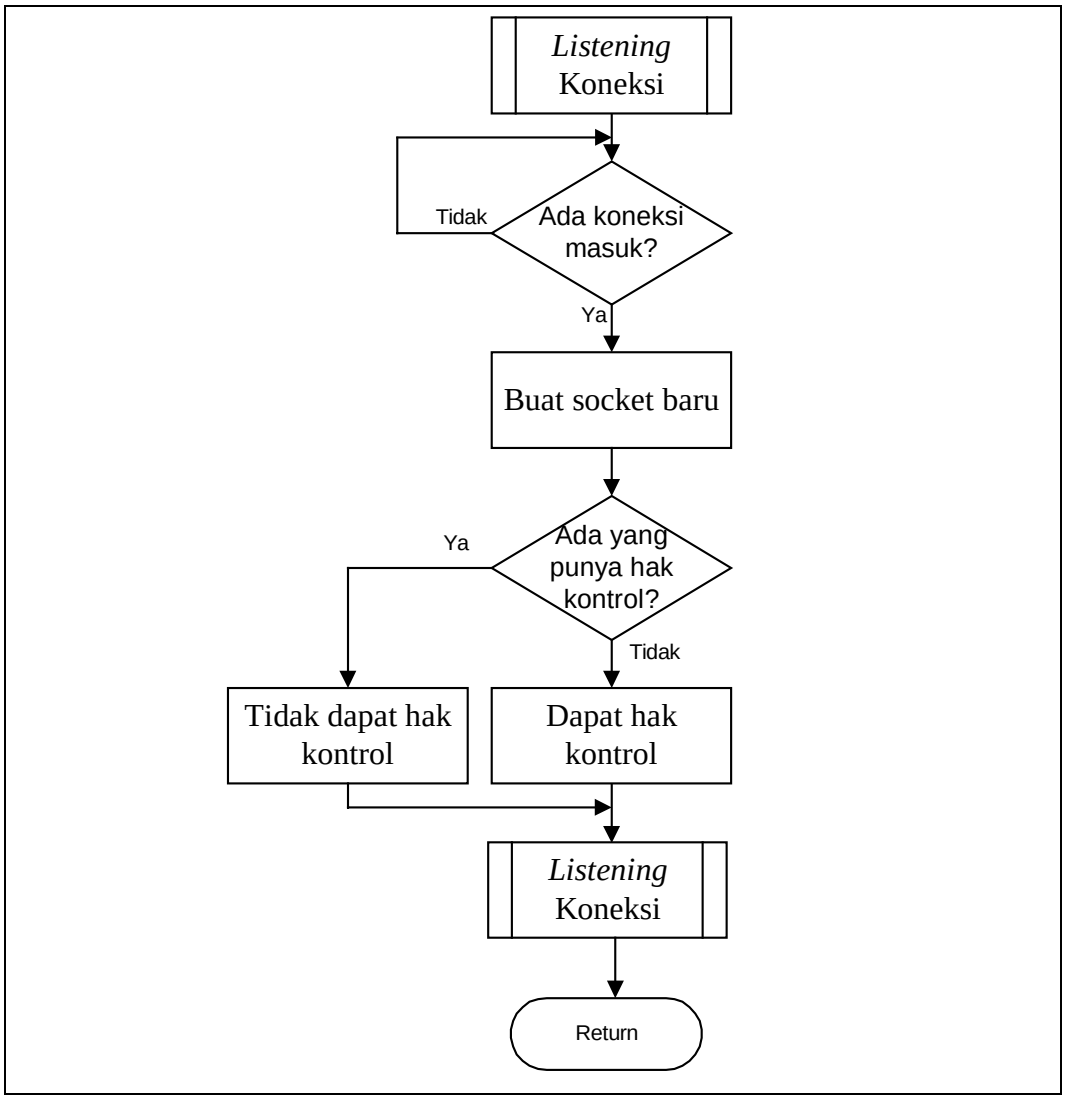
Langkah selanjutnya adalah melakukan pemanggilan terhadap fungsi *listen* yang akan membuat *socket* sListen menunggu koneksi dari *client*. Nilai 3 pada parameter ke-2 fungsi ini menunjukkan jumlah antrian yang bisa ditampung fungsi *listen*. Jika fungsi *listen* gagal melakukan tugasnya maka akan ditampilkan pesan “Server could not listen!”. Berikut ini potongan kode programnya:

```
if ( listen(sListen, 3) == SOCKET_ERROR )
{
      MessageBox("Server could not listen!", "Server",
      MB_OK+MB_ICONERROR);
      return FALSE;
}
```

3.1.1.2. Prosedur *Listening* Koneksi

Kalau pada bagian sebelumnya, telah dibahas mengenai pemanggilan terhadap fungsi *listen* untuk menunggu koneksi dari *client*, maka prosedur *listening* koneksi ini melakukan hal-hal yang berhubungan dengan bagaimana fungsi *accept* menerima koneksi yang datang dari *client* dan membuat *socket* yang baru untuk menampung koneksi tersebut. Juga dibahas mengenai mekanisme untuk menentukan berhak atau tidaknya *client* tersebut mendapatkan hak kontrol.

Diagram alur untuk prosedur ini adalah sebagai berikut:



Gambar 3.6. Diagram Alur Prosedur *Listening* Koneksi

Pada bagian sebelumnya, telah dibahas juga mengenai pemanggilan terhadap fungsi WSAAsyncSelect yang mendaftarkan sebuah fungsi yang akan menangani *event-event* dan *message-message* yang berhubungan dengan jaringan. Fungsi WSAAsyncSelect tersebut akan dibahas lebih mendalam lagi pada bagian ini.

Nilai WM_USER_ASYNC_SELECT yang dipakai sebagai parameter dalam fungsi WSAAsyncSelect dideklarasikan dalam file ServerDlg.h. Berikut ini potongan kode yang mendeklarasikan nilai tersebut:

```
#define WM_USER_ASYNC_SELECT (WM_USER + 1)
```

Sedangkan fungsi yang akan menangani *event-event* yang berhubungan dengan jaringan dideklarasikan pada tingkat *protected* kelas CServerDlg. Fungsi tersebut bernama OnAsyncSelect. Berikut ini potongan kode program yang mendeklarasikan fungsi tersebut:

```
class CServerDlg : public CDialog
{
// Construction
public:
	. . .
protected:
	. . .
	afx_msg LONG OnAsyncSelect(WPARAM wParam, LPARAM lParam);
	. . .
};
```

Untuk membuat fungsi OnAsyncSelect bisa terpanggil secara otomatis bila terjadi *message* WM_USER_ASYNC_SELECT maka perlu mendaftarkan *message* tersebut pada *message map* aplikasi. Potongan kode program berikut terdapat dalam file ServerDlg.cpp akan menjelaskan bagaimana mendaftarkan *message* WM_USER_ASYNC_SELECT pada aplikasi:

```
BEGIN_MESSAGE_MAP(CServerDlg, CDialog)
	//{{AFX_MSG_MAP(CServerDlg)
	. . .
	ON_MESSAGE(WM_USER_ASYNC_SELECT, OnAsyncSelect)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Fungsi OnAsyncSelect akan merespon semua *event-event* yang terjadi pada jaringan. Dalam fungsi ini, akan dipilih *event* apa yang telah terjadi untuk kemudian dilakukan respon terhadap *event* tersebut. Ada 2 *event* yang direspon di sini, FD_ACCEPT dan FD_READ. Berikut ini potongan kode dari fungsi tersebut:

```
LONG CServerDlg::OnAsyncSelect(WPARAM wParam,LPARAM lParam)
{
	if ( WSAGETSELECTERROR(lParam) != 0 )
		return 0L;
	switch ( WSAGETSELECTEVENT(lParam) )
	{
		case FD_ACCEPT :
			OnAccept();
			break;
		case FD_READ:
			OnReceiveData();
			break;
	}
	return 0L;
}
```

Jika ada koneksi baru dari *client* yang masuk ke *socket server* yang sedang melakukan proses *listening* (menunggu koneksi yang masuk) maka fungsi WSAGETSELECTEVENT yang menerima masukkan parameter lParam akan menghasilkan nilai FD_ACCEPT. Dari potongan kode di atas, diketahui bahwa jika pernyataan *switch* ternyata menjukkan pilihan FD_ACCEPT maka akan dilakukan pemanggilan terhadap fungsi OnAccept. Berikut ini potongan kode dari fungsi OnAccept:

```
for (int i = 0; i < 10; i++)
{
      if ( sockClient[i] == INVALID_SOCKET &&
      find == FALSE)
      {
            sockClient[i] = accept(sListen,
            (struct sockaddr*)&saClient, &size);

            if ( sockClient[i] == INVALID_SOCKET )
                  MessageBox("Socket ERROR!", "Server",
                  MB_OK+MB_ICONERROR);
            else
            {
                  find = TRUE;
                  . . .
            }
      }
}
```

Potongan kode program di atas dapat dijelaskan sebagai berikut: *Socket* sockClient yang digunakan untuk menangani koneksi dengan *client* yang baru, dipilih dari *socket* sockClient yang sedang tidak digunakan oleh *client* yang lain. Pencarian *socket* sockClient tersebut dilakukan dalam perulangan berikut:

```
for (int i = 0; i < 10; i++)
{
      if ( sockClient[i] == INVALID_SOCKET && . . .)
      {
            . . .
      }
}
```

Koneksi yang masuk ke *socket server* ditangani oleh fungsi *accept*. Jika fungsi *accept* gagal menerima koneksi dari *client* maka akan menampilkan pesan “Socket ERROR!”. Berikut ini potongan kode untuk menangani pemanggilan fungsi *accept*:

```
sockClient[i] = accept(sListen,
                  (struct sockaddr*)&saClient, &size);
if ( sockClient[i] == INVALID_SOCKET )
      MessageBox("Socket ERROR!", "Server", MB_OK+MB_ICONERROR);
```

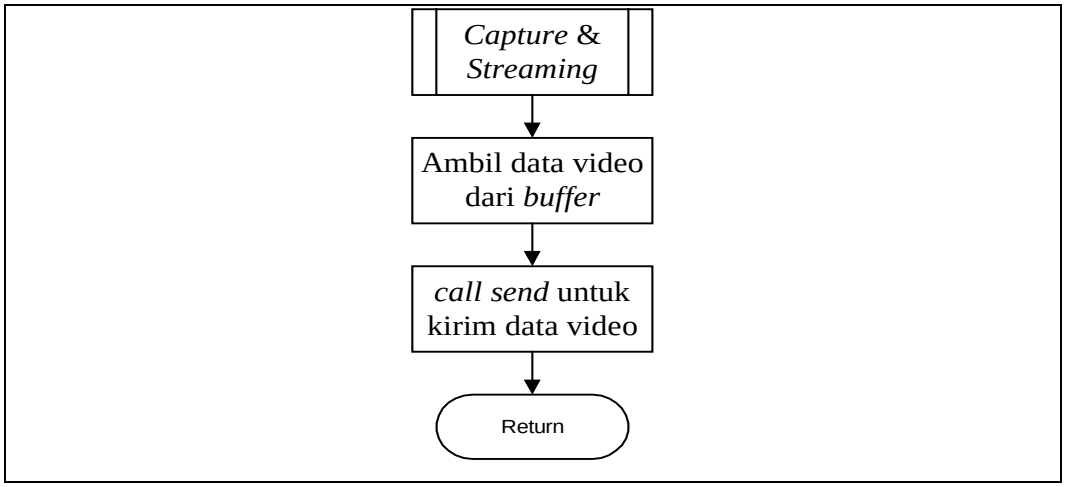
Variabel whoControl digunakan untuk mengetahui *client* mana yang sedang memegang hak kontrol. Jika variabel ini bernilai –1 maka pada saat itu tidak ada *client* yang sedang memegang hak kontrol. Berikut ini potongan kodenya:

```
if ( sockClient[i] == INVALID_SOCKET )
      MessageBox("Socket ERROR!", "Server", MB_OK+MB_ICONERROR);
else
{
      find = TRUE;
      if ( whoControl == -1 )
      {
            whoControl = i;
            send(sockClient[i],buff1,8,0);
      }
      else
            send(sockClient[i],buff2,8,0);
}
```

3.1.1.3. Prosedur *Capture* dan *Streaming*

Pada prosedur ini, akan dibahas mengenai proses *capture* dari *webcam* dan mekanisme pengiriman data video hasil *capture* dari komputer *server* ke komputer *client*. Proses *capture* mencakup inisialisasi dan pemanggilan fungsi *callback* untuk mendapatkan *buffer* data video sedangkan mekanisme pengiriman data video hasil *capture* dari *server* ke *client* mencakup pemanggilan fungsi *send* milik winsock untuk mengirimkan data video tersebut ke *client*.

Berikut ini diagram alur untuk prosedur *capture* dan *streaming*:



Gambar 3.7. Diagram Alur Prosedur *Capture* dan *Streaming*

Supaya aplikasi *server* mampu mengirimkan data video ke *client* secara *streaming*, maka aplikasi *server* perlu mendapatkan data video secara langsung dari *buffer memory*. Data video yang diperoleh dari *buffer memory* tersebut dikirimkan ke komputer *client* dengan memanggil fungsi *send*.

Untuk bisa mendapatkan data video dari *buffer memory*, aplikasi *server* harus melakukan inisialisasi sebuah fungsi *callback* pada aplikasi *server* yang mana fungsi *callback* tersebut diberi nama capVideoStreamCallback. Fungsi yang digunakan untuk menginisialisasi fungsi *callback* tersebut adalah fungsi SetStreamCallBack milik kelas AviCap. Fungsi SetStreamCallBack tersebut melakukan pemanggilan terhadap *macro* capSetCallbackOnVideoStream milik Windows API. Berikut potongan kode program untuk menginisialisasi fungsi *callback* capVideoStreamCallback:

```
capwnd.SetStreamCallBack(capVideoStreamCallback);
```

Sedangkan fungsi *callback* capVideoStreamCallback dideklarasikan dalam file ServerDlg.h. Berikut potongan kode program yang mendeklarasikan fungsi *callback* tersebut:

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

Setelah mendeklarasikan fungsi *callback* untuk memperoleh data video dari *buffer memory* secara langsung, langkah berikutnya adalah memulai proses pengambilan data video dengan pemanggilan fungsi StartSeq milik kelas AviCap. Fungsi StartSeq melakukan pemanggilan terhadap *macro* milik Windows API yaitu *macro* capCaptureSequenceNoFile. Potongan kode program berikut ini akan menjelaskannya:

```
capwnd.StartSeq(TRUE);
```

Setelah pemanggilan fungsi StartSeq milik kelas AviCap tersebut maka setiap kali *buffer memory video* sudah terisi dengan data video, akan dilakukan pemanggilan terhadap fungsi *callback* capVideoStreamCallback yang telah diinisialisasi sebelumnya. *Buffer* data video dapat ditemukan dalam struktur LPVIDEOHDR pada *member* lpData yang bertipe LPBYTE. Berikut ini potongan kode program fungsi *callback* capVideoStreamCallback:

```
LRESULT CALLBACK capVideoStreamCallback(HWND hWnd, LPVIDEOHDR
lpVHdr)
{
      . . .
      lData = lpVHdr->lpData;
      . . .
      return 0L;
}
```

Supaya data video yang diperoleh dari *buffer memory* bisa dikirim ke komputer *client* maka data video tersebut harus diubah dulu dari bentuk *byte* ke bentuk *char*. Pengubahan ini ternyata menimbulkan masalah jika ada nilai ASCII 0 (nol) yang merupakan *null-terminated*. Akibat yang ditimbulkan *null-terminated* tersebut adalah data video yang akan dikirimkan menjadi terpotong pada titik tersebut. Oleh karena itu, nilai ASCII 0 pada data video harus diubah menjadi nilai ASCII 1 untuk menghindari pemotongan tersebut, hal ini dengan yakin dilakukan karena intensitas warna yang dihasilkan nilai ASCII 0 dan nilai ASCII 1 tidak akan tampak berbeda. Berikut ini potongan kode programnya:

```
lData = lpVHdr->lpData;
for(int u = 0; u < 57600; u++)
{
      if ( *(lData+u) == 0 )
            *(lData+u) = 1;
}
```

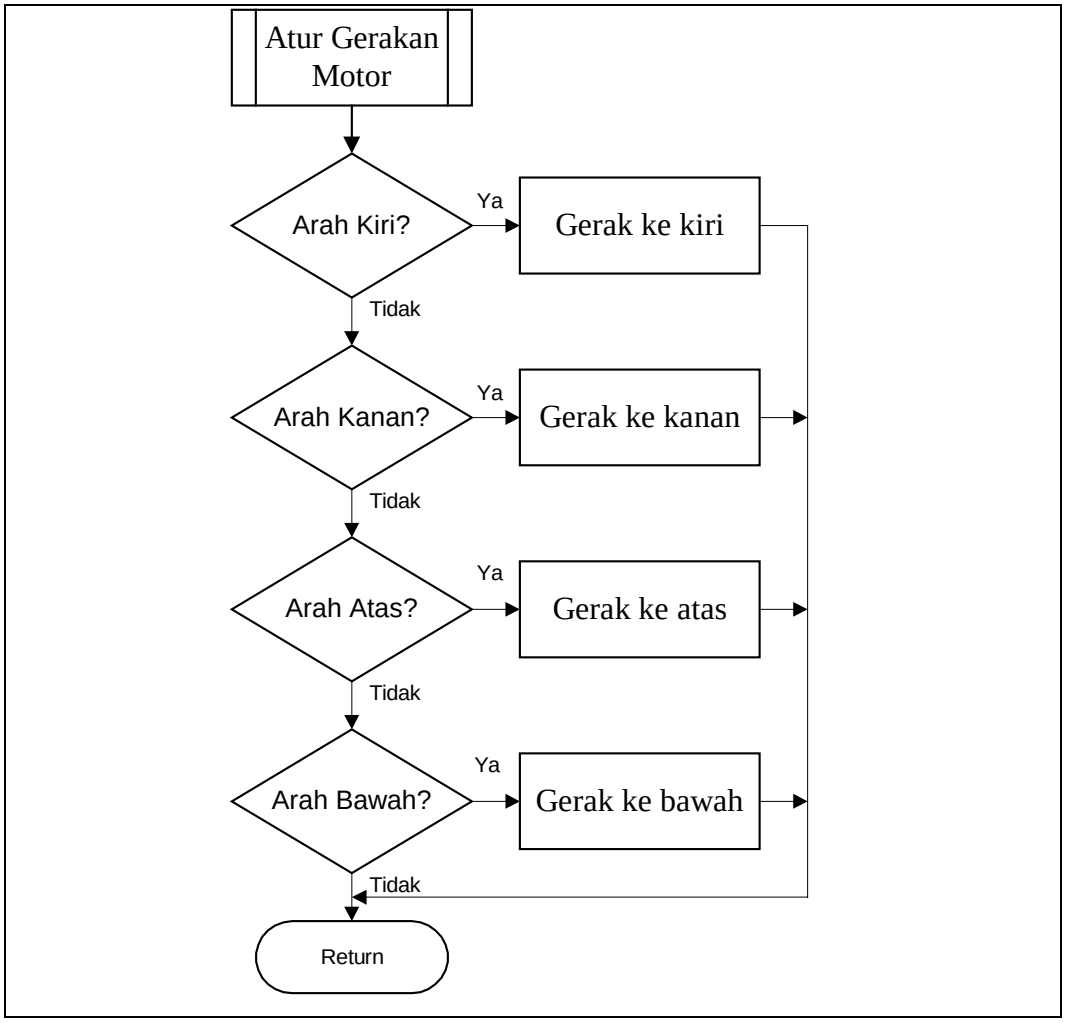
Setelah data video siap tibalah waktunya data tersebut dikirim ke *client* dengan memanggil fungsi *send* milik winsock. Supaya pada saat pengiriman data ke *client* tidak mengganggu proses lain yang sedang berjalan pada aplikasi *server* maka dibuat sebuah *thread* baru yang di mana fungsi *send* berjalan dalam *thread* baru tersebut. Berikut potongan kode programnya:

```
UINT transmite(LPVOID p)
{
      transmiteOK = 1;
      CServerDlg* dlg = (CServerDlg*)AfxGetApp()->GetMainWnd();
      char *buff = (char*)lData;
      for (int i = 0; i < 10; i++)
      {
            if ( (dlg->sockClient[i] != INVALID_SOCKET) &&
            (stream[i] == 1) )
            {
                  send(dlg->sockClient[i],buff,57600,0);
                  Sleep(1);
            }
      }
      transmiteOK = 0;
      return 0;
}
```

3.1.1.4. Prosedur Atur Gerakan Motor

Selain bisa mengirimkan data video ke komputer *client*, komputer *server* juga mampu menerima perintah dari komputer *client* (dalam hal ini komputer *client* yang memiliki hak kontrol) untuk menggerakkan kamera. Perintah yang diterima dari *client* diteruskan ke perangkat penggerak kamera melalui paralel *port*.

Berikut ini diagram alur untuk prosedur Atur Gerakan Motor:



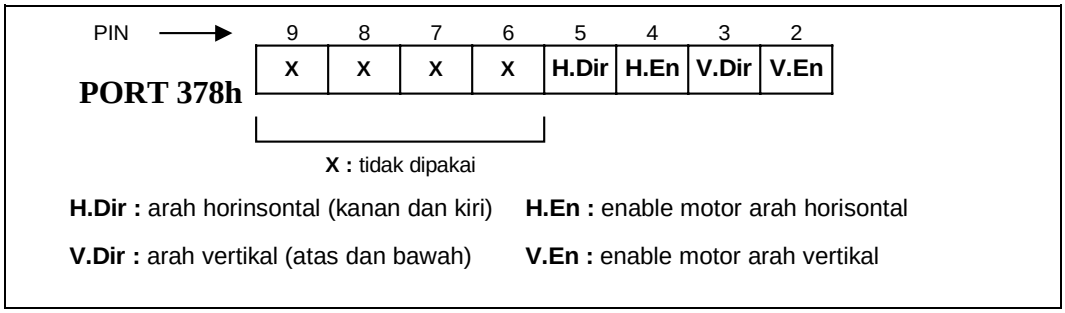
Gambar 3.8. Diagram Alur ProsedurAtur Gerakan Motor

Fungsi *recv* milik winsock berfungsi untuk menerima pesan yang masuk ke *socket server*. Pesan yang masuk ke *server* dapat digolongkan menjadi dua macam, yaitu pesan yang berisi perintah untuk menggerakkan kamera dan pesan yang berisi perintah untuk status *server*. Dalam prosedur ini, pesan yang dibahas

adalah pesan yang berisi perintah untuk menggerakkan kamera. Berikut ini potongan kode pemanggilan terhadap fungsi *recv* yang terdapat dalam fungsi OnReceiverData milik kelas CServerDlg:

```
char buff[256];
. . .
for (int i = 0; i < 10; i++)
{
      if ( sockClient[i] != INVALID_SOCKET )
      {
            int nbyte = recv(sockClient[i],buff,255,0);
            . . .
      }
}
```

Pesan yang berisi perintah untuk menggerakkan kamera tersebut terdiri dari empat macam, yaitu untuk arah kiri, kanan, atas dan bawah. Gambar berikut menunjukkan disain keluaran pada pin-pin paralel *port* (alamat 378h) untuk masing-masing arah tersebut:



Gambar 3.9. Konfigurasi Keluaran *Port* 378h

Kedua pin *enable* (H.En dan V.En) merupakan pin-pin dengan logika *active-low*. Pin ke-4 (H.En) digunakan untuk menghidupkan dan mematikan motor horisontal (motor yang berfungsi menggerakkan kamera ke kanan dan ke kiri). Jika pin ini diberi logika *high* maka motor akan berada dalam keadaan mati, sedangkan jika pin ini diberi logika *low* maka motor akan berputar sesuai arah yang ditunjukkan pin ke-5 (H.Dir).

Pin ke-2 (V.En) digunakan untuk menghidupkan dan mematikan motor vertikal (motor yang berfungsi menggerakkan kamera ke atas dan ke bawah). Jika pin ini diberi logika *high* maka motor akan berada dalam keadaan mati, namun sebaliknya jika diberi logika *low* maka motor akan berputar sesuai arah yang ditunjukkan pin ke-3 (V.Dir).

Pin ke-5 (H.Dir) digunakan untuk menentukan arah putaran motor horisontal. Jika ingin menggerakkan kamera ke arah kanan maka pin ke-5 ini diberi logika *high* namun jika menginginkan gerakan kamera ke arah kiri, pin ke-5 ini diberi logika *low*.

Jika pin ke-5 digunakan untuk menentukan arah putaran motor horisontal maka pin ke-3 (V.Dir) digunakan untuk menentukan arah putaran motor vertikal. Jika ingin menggerakkan kamera ke arah atas maka pin ke-3 ini diberi logika *low* namun jika menginginkan gerakan kamera ke arah bawah, pin ke-3 ini diberi logika *high*.

Selain mengeluarkan sinyal-sinyal ke paralel *port* pada alamat 378h, aplikasi *server* juga membaca sinyal-sinyal yang masuk pada alamat 379h. Sinyal-sinyal ini berasal dari *limit switch* yang ada ada penggerak kamera, yang berfungsi untuk membatasi putaran motor vertikal. Jika tuas pada motor vertikal sudah menyentuh *switch* tersebut sehingga *switch* tertekan maka melalui *port* 379h inilah sinyal-sinyal tertentu dikirimkan ke komputer *server* dan aplikasi *server* dengan segera mengirim sinyal balasan melalui *port* 378h untuk mematikan motor vertikal. Gambar berikut ini menunjukkan disain masukan pada pin-pin paralel *port* (alamat 379h):

PIN →	11	10	12	13	15				nilai dalam desimal
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	↓
Switch depan tertekan	0	1	0	1	0	1	1	0	86
Switch belakang tertekan	0	1	1	0	0	1	1	0	102
Switch tidak tertekan	0	1	1	1	0	1	1	0	118

Gambar 3.10. Konfigurasi Masukan *Port* 379h

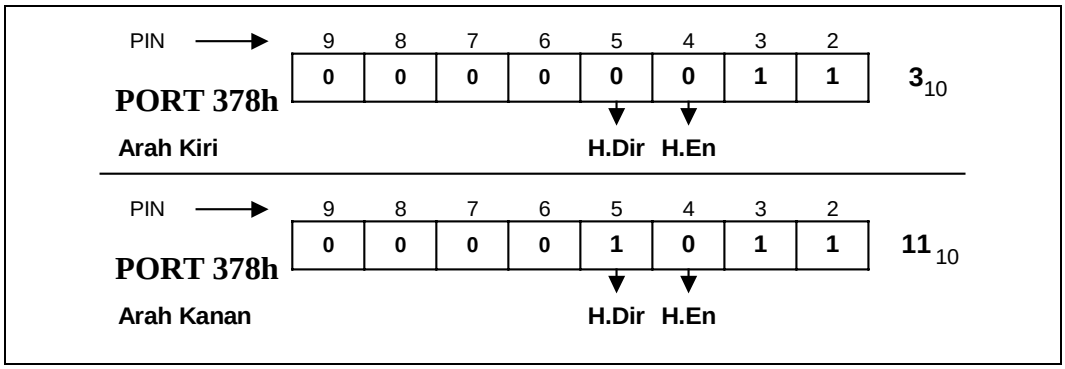
Pin ke-10 dan pin ke-11 (*inverted*) tidak digunakan namun pin-pin tersebut di-*pull-up*, sehingga pin-pin tersebut tetap menghasilkan sinyal meskipun tidak berpengaruh pada sistem. Karena pin ke-11 merupakan pin *inverted* maka pin ini tidak menghasilkan logika *high* melainkan logika *low*. Sedangkan pin ke-15 dihubungkan ke *ground* sehingga menghasikan logika *low*. Pin ke-12

terhubung dengan *switch* depan, sedangkan pin ke-13 terhubung dengan *switch* belakang.

Berdasarkan gambar di atas, prinsip kerja masukan *port* 379h tersebut dapat dijelaskan sebagai berikut: dalam keadaan normal (kedua *switch* tidak dalam keadaan tertekan), *port* 379h ini memberi masukan nilai 118_{10} pada komputer *server*. Dalam keadaan yang disebut normal ini, semua motor baik motor horisontal maupun motor vertikal dapat diaktifkan. Pin ke-12 dan pin ke-13 merupakan kunci utama pada *port* 379h ini. Dalam keadaan normal inilah, kedua pin tersebut memberikan sinyal dengan logika *high*, karena rangkaian bekerja dengan logika *active-low* (jika *switch* tertekan baru menghasilkan logika *low*) berarti tidak ada *switch* yang tertekan.

Jika kemudian *switch* depan tertekan akibat putaran motor vertikal telah mencapai titik batasnya maka pin ke-12 menjadi *low* sedangkan pin ke-13 tetap *high*, dan *port* 379h memberikan masukan nilai 86_{10} pada komputer *server*. Keadaan ini mencegah aplikasi *server* menghidupkan motor vertikal.

Demikian juga sebaliknya, jika *switch* belakang tertekan akibat putaran motor vertikal telah mencapai titik batasnya maka pin ke-13 menjadi *low* sedangkan pin ke-12 tetap *high*, dan *port* 379h memberikan masukan nilai 102_{10} pada komputer *server*. Keadaan ini juga mencegah aplikasi *server* menghidupkan motor vertikal.



Gambar 3.11. Konfigurasi Keluaran untuk Putaran ke Kiri dan Kanan

Seperti apa yang sudah dibahas di bagian sebelumnya, putaran motor ke kanan dan ke kiri dilakukan dengan mengatur pin ke-4 (*enable*) dan pin ke-5 (arah putaran). Gambar 3.11 menunjukkan konfigurasi pin-pin pada *port* 378h ketika

aplikasi *server* memberikan perintah kepada *driver* penggerak kamera untuk menggerakkan kamera ke kanan dan ke kiri. Pengkodeannya dapat dilihat pada potongan kode berikut:

```
char buff[256];
char aLeft[256] = "KIRI";
char aRight[256] = "KANAN";
. . .
if ( i == whoControl )
{
      . . .
      if (strcmp(buff,aLeft) == 0)
      {
            __asm mov edx,0x378
            __asm mov al,3
            __asm out dx,al
            Sleep(40);
            __asm mov edx,0x378
            __asm mov al,255
            __asm out dx,al
      }
      if (strcmp(buff,aRight) == 0)
      {
            __asm mov edx,0x378
            __asm mov al,11
            __asm out dx,al
            Sleep(40);
            __asm mov edx,0x378
            __asm mov al,255
            __asm out dx,al
      }
      . . .
}
```

Mula-mula dilakukan pembandingan variabel buff dengan aLeft (untuk arah kiri) dan buff dengan aRight (untuk arah kanan) dengan menggunakan fungsi strcmp. Jika ternyata isi variabel buff sama dengan isi variabel aLeft maka nilai 3_{10} diberikan pada *port* 378h supaya motor horisontal berputar ke kiri.

```
__asm mov edx,0x378
__asm mov al,3
__asm out dx,al
```

Pemanggilan fungsi Sleep dengan menyertakan parameter 40 berguna untuk membuat *delay* selama 40ms. *Delay* ini yang menentukan besarnya sudut putaran motor tiap kali motor diperintahkan untuk bergerak. Semakin besar nilai yang disertakan pada fungsi Sleep maka semakin besar sudut putaran motor.

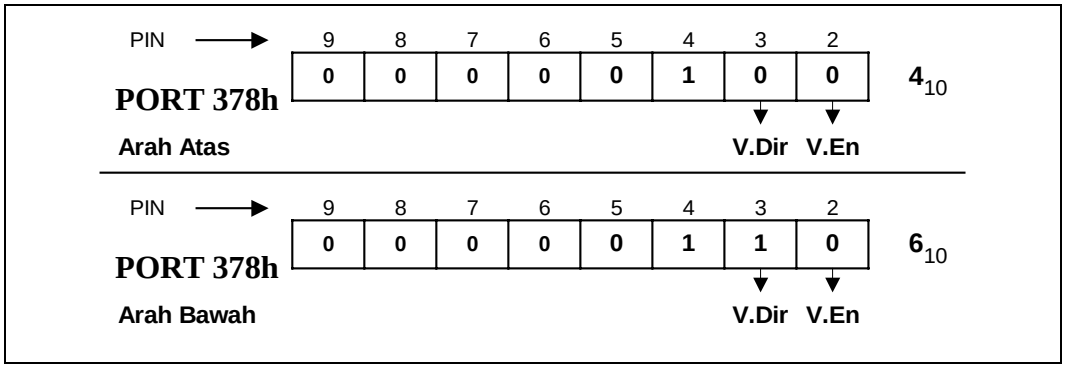
Ketika suatu nilai diberikan pada paralel *port*, nilai tersebut akan terus dipertahankan sampai ada nilai baru yang diberikan padanya. Hal ini akan menjadi masalah yang cukup berarti, misalnya jika aplikasi memberikan nilai 3_{10}

pada *port* 378h (untuk memerintahkan motor horisontal berputar ke kiri), maka nilai tersebut akan membuat motor horisontal terus berputar ke kiri dan tidak akan berhenti sebelum mencapai batas putarannya. Tentunya hal tersebut sangat tidak diinginkan, dan untuk mencegah motor horisontal terus berputar ke kiri maka *port* 378h harus diberi nilai 255_{10}.

```
__asm mov edx,0x378
__asm mov al,255
__asm out dx,al
```

Hal yang sama juga berlaku jika isi variabel buff sama dengan isi variabel aRight, hanya saja nilai yang diberikan pada *port* 378h adalah 11_{10}.

```
__asm mov edx,0x378
__asm mov al,11
__asm out dx,al
```



Gambar 3.12. Konfigurasi Keluaran untuk Putaran ke Atas dan Bawah

Gambar 3.12 menunjukkan konfigurasi pin-pin pada *port* 378h ketika aplikasi *server* memberikan perintah kepada *driver* penggerak kamera untuk menggerakkan kamera ke atas dan ke bawah. Pengkodeannya dapat dilihat pada potongan kode berikut:

```
char buff[256];
char aUp[256] = "ATAS";
char aDown[256] = "BAWAH";
. . .
if ( i == whoControl )
{
      . . .
      if (strcmp(buff,aUp) == 0)
      {
            __asm mov edx,0x379
            __asm in al,dx
            __asm mov val,al

            if (val == 118 || val == 86)
```

```
            {
                  __asm mov edx,0x378
                  __asm mov al,4
                  __asm out dx,al
                  Sleep(40);
                  __asm mov edx,0x378
                  __asm mov al,255
                  __asm out dx,al
            }
      }

      if (strcmp(buff,aDown) == 0)
      {
            __asm mov edx,0x379
            __asm in al,dx
            __asm mov val,al

            if (val == 118 || val == 102)
            {
                  __asm mov edx,0x378
                  __asm mov al,6
                  __asm out dx,al
                  Sleep(40);
                  __asm mov edx,0x378
                  __asm mov al,255
                  __asm out dx,al
            }
      }
      . . .
}
```

Langkah-langkahnya hampir sama dengan bagian sebelumnya, hanya saja pada pengaturan putaran motor vertikal ini terdapat tambahan beberapa baris kode. Tambahan kode tersebut berhubungan dengan masukan pada *port* 379h yang memperoleh sinyal dari *switch* yang terpasang pada alat penggerak kamera (lihat gambar 3.10). Mula-mula dilakukan pembandingan variabel buff dengan aUp (untuk arah atas) dan aDown (untuk arah bawah) dengan menggunakan fungsi strcmp. Jika ternyata isi variabel buff sama dengan isi variabel aUp maka langkah berikutnya adalah membaca nilai masukan yang ada pada *port* 379h.

```
unsigned char val;
. . .
if (strcmp(buff,aUp) == 0)
{
      __asm mov edx,0x379
      __asm in al,dx
      __asm mov val,al
      . . .
}
```

Nilai yang diperoleh dari pembacaan *port* 379h disimpan ke dalam variabel val. Nilai variabel val inilah yang menentukan perputaran motor vertikal,

artinya motor vertikal akan berputar ke atas jika kondisi yang mengijinkan motor berputar terpenuhi (lihat gambar 3.10).

```
if (strcmp(buff,aUp) == 0)
{
      __asm mov edx,0x379
      __asm in al,dx
      __asm mov val,al
      if (val == 118 || val == 86)
      {
            __asm mov edx,0x378
            __asm mov al,4
            __asm out dx,al
            Sleep(40);
            __asm mov edx,0x378
            __asm mov al,255
            __asm out dx,al
      }
}
```

Pada potongan kode di atas, jika nilai variabel val sama dengan 118 (berarti tidak ada *switch* yang tertekan) atau nilai variabel val sama dengan 86 (berarti *swicth* depan dalam keadaan tertekan) maka motor vertikal dapat digerakkan ke atas. Motor hanya bisa berputar jika salah satu dari dua kondisi di atas terpenuhi, dengan pengaturan seperti ini menghindarkan motor terus berputar meskipun sudah tidak dapat beranjak dari suatu posisi (posisi menekan *switch* belakang).

Untuk putaran ke bawah, hal yang sama dengan putaran ke atas berlaku juga. Jika ternyata isi variabel buff sama dengan isi variabel aDown maka langkah berikutnya adalah membaca nilai masukan yang ada pada *port* 379h.

```
unsigned char val;
. . .
if (strcmp(buff,aDown) == 0)
{
      __asm mov edx,0x379
      __asm in al,dx
      __asm mov val,al
      . . .
}
```

Nilai yang diperoleh dari pembacaan *port* 379h disimpan ke dalam variabel val. Nilai variabel val inilah yang menentukan perputaran motor vertikal, artinya motor vertikal akan berputar ke bawah jika kondisi yang mengijinkan motor berputar terpenuhi (lihat gambar 3.10).

```
if (strcmp(buff,aDown) == 0)
{
      __asm mov edx,0x379
      __asm in al,dx
      __asm mov val,al
      if (val == 118 || val == 102)
      {
            __asm mov edx,0x378
            __asm mov al,6
            __asm out dx,al
            Sleep(40);
            __asm mov edx,0x378
            __asm mov al,255
            __asm out dx,al
      }
}
```

Pada potongan kode di atas, jika nilai variabel val sama dengan 118 (berarti tidak ada *switch* yang tertekan) atau nilai variabel val sama dengan 102 (berarti *swicth* belakang dalam keadaan tertekan) maka motor vertikal dapat digerakkan ke bawah. Motor hanya bisa berputar jika salah satu dari dua kondisi di atas terpenuhi, dengan pengaturan seperti ini menghindarkan motor terus berputar meskipun sudah tidak dapat beranjak dari suatu posisi (posisi menekan *switch* depan).

3.1.2. Perancangan dan Implementasi Aplikasi *Client*

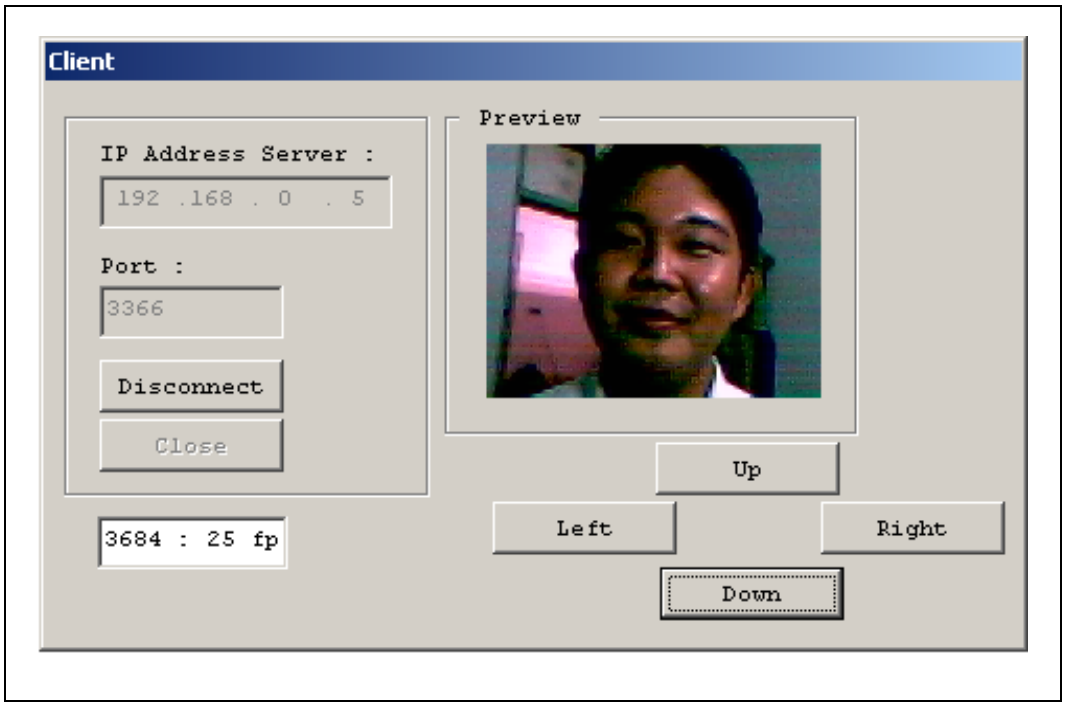
Aplikasi *client* dirancang untuk melakukan hal-hal sebagai berikut:

- Membuat *socket* dan melakukan usaha koneksi ke komputer *server*.
- Menerima data video dari komputer *server*.
- Mengubah data video yang diterima dari komputer *server* menjadi gambar dan menampilkannya ke layar.
- Mengirimkan pesan yang berisi perintah untuk menggerakkan kamera ke komputer *server*.
- Memutuskan koneksi dengan *server*.

Pemrograman *socket* dilakukan dengan menggunakan winsock dalam bahasa C++ dengan kompiler yang dipakai adalah Microsoft Visual C++ versi 6. Aplikasi ditulis dengan bantuan fasilitas *library* Microsoft Foundation Class (MFC), yang mempermudah dalam pembuatan *user interface* untuk aplikasi. Jenis protokol yang dipakai adalah TCP karena jenis koneksi yang digunakan adalah *connection-oriented*.

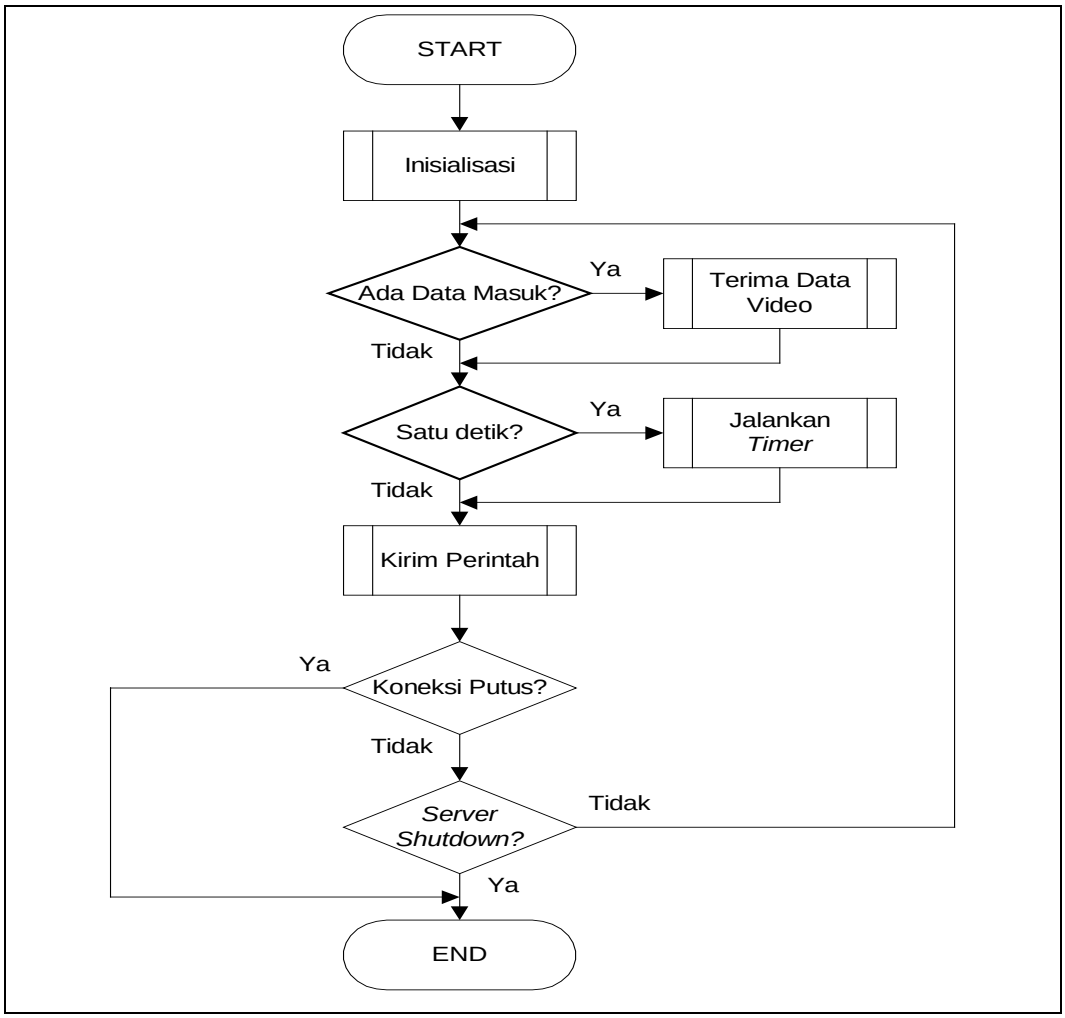
Untuk menampilkan gambar ke layar, aplikasi *client* menggunakan fungsi-fungsi dari *library* DrawDib (lihat sub bab 2.3 yang membahas tentang DrawDib). Selain menampilkan gambar tiap *frame* ke layar, aplikasi *client* juga menghitung dan menampilkan jumlah *frame* yang berhasil diterima.

Pembuatan aplikasi *client* jauh lebih sederhana dibandingkan dengan pembuatan aplikasi *server*. Tugas yang ada pada aplikasi *client* juga lebih sedikit dari tugas yang ada pada aplikasi *server*. Aplikasi *client* hanya mengurus koneksi dengan aplikasi *server*. Tidak seperti aplikasi *server* yang selain mengurus koneksi dengan *client*, juga mengurus koneksi dengan *webcam*.



Gambar 3.13. *Screen Shot* Aplikasi *Client*

Sistem kerja aplikasi *client* akan dijelaskan dalam diagram alur di bawah ini:



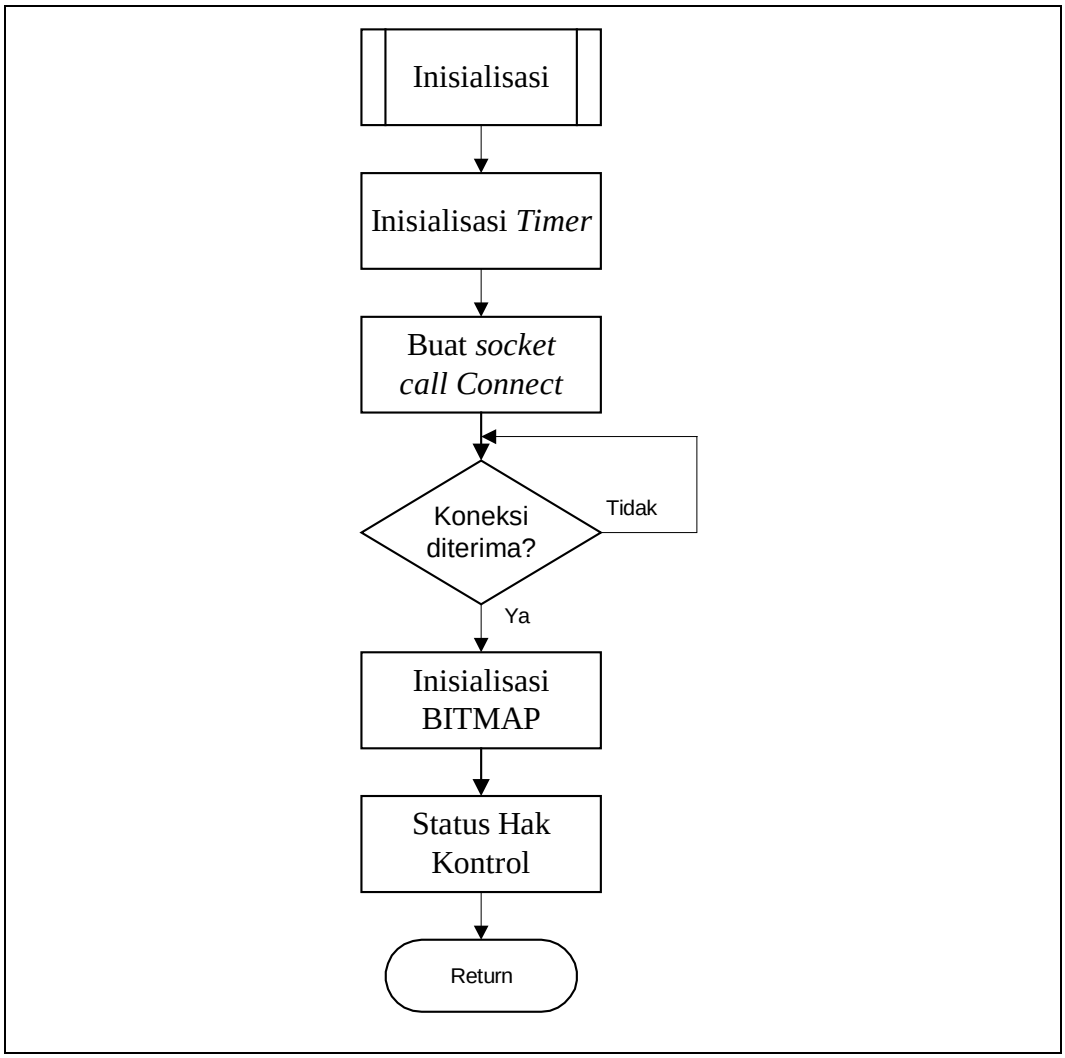
Gambar 3.14. Diagram Alur Aplikasi *Client*

Diagram alur aplikasi *client* di atas menggambarkan alur kerja dari aplikasi *cilent*. Di dalam diagram alur tersebut terdapat beberapa prosedur, seperti inisialisasi, jalankan *timer*, kirim perintah dan terima data video. Di dalam tiap-tiap prosedur tersebut terdapat proses-proses, yang menjelaskan hal-hal yang terjadi dalam aplikasi. Berikut ini penjelasan proses-proses yang terjadi dalam tiap-tiap prosedur:

3.1.2.1. Prosedur Inisialisasi

Prosedur inisialisasi ini melakukan hal-hal yang berhubungan dengan pembuatan *socket client*, inisialisasi *timer*, status hak kontrol dan inisialisasi struktur BITMAPINFOHEADER.

Diagram alur untuk prosedur ini adalah sebagai berikut:



Gambar 3.15. Diagram Alur Prosedur Inisialisasi Aplikasi *Client*

Timer di sini digunakan untuk mengarahkan program menuju ke suatu proses atau fungsi setiap satu detik sekali. Tujuan dari penggunaan *timer* ini adalah untuk menampilkan jumlah *frame* yang berhasil diterima oleh komputer *client*. Setiap kali aplikasi *client* menerima satu *frame* maka dilakukan penambahan terhadap suatu veriabel tertentu. Setiap satu detik, nilai variabel tersebut ditampilkan dan setelah itu nilai variabel dinolkan kembali.

Untuk keperluan menginisialisasi *timer* maka digunakan fungsi SetTimer. Parameter kedua menyatakan interval waktu, yang menentukan setiap

berapa lama sekali program akan mengarah ke sebuah fungsi yang telah didefinisikan. Berikut ini potongan kode programnya:

```
SetTimer(2,1000,NULL);
```

Langkah berikutnya adalah membuat sebuah *socket* yang akan digunakan untuk melakukan koneksi dengan aplikasi *server*. Pada aplikasi *client*, hanya ada sebuah *socket* beserta strukturnya. *Socket* yang akan dibuat bernama sClient sedangkan struktur SOCKADDR_IN milik *socket* sClient bernama saClient. *Socket* sClient dan struktur SOCKADDR_IN saClient dideklarasikan pada kelas CClientDlg pada file ClientDlg.h.

```
class CClientDlg : public CDialog
{
public:
      . . .
      SOCKET sClient;
      SOCKADDR_IN saClient;
      . . .
}
```

Setelah *socket* sClient dan struktur SOCKADDR_IN saClient selesai dideklarasikan, maka *socket* perlu segera dibuat. Pembuatan *socket* sClient juga menggunakan fungsi *socket* milik winsock. Parameter AF_INET pada fungsi tersebut merupakan alamat *family*. Sedangkan parameter SOCK_STREAM dipakai di sini karena jenis protokol yang digunakan adalah TCP. Jika *socket* gagal diciptakan maka akan ditampilkan pesan “Failed to create socket!”.

```
sClient = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if ( sClient == INVALID_SOCKET )
{
      MessageBox("Failed to create socket!", "Client",
      MB_OK+MB_ICONERROR);
      return FALSE;
}
```

Bagian ini akan menjelaskan pengisian nilai pada struktur saClient serta bagaimana melakukan koneksi ke *server* dengan memanggil fungsi *connect*. Nomor *port* dari *server* yang ada pada *member variable* m_port dimasukkan sebagai nilai dari *member* sin_port. Alamat IP *server* yang dituju dapat diperoleh pada variabel s yang bertipe CString, yang mengambil alamat IP *server* dari kontrol edit. Pengisian nilai pada *member* struktur saClient berguna pada langkah selanjutnya, yaitu pada pemanggilan fungsi *connect*.

Pemanggilan fungsi *connect* ini akan menyebabkan *socket* sClient berusaha melakukan koneksi ke *server* pada alamat dan nomor *port* yang telah ditentukan oleh struktur saClient. Jika *socket* gagal melakukan koneksi dengan *server* maka aplikasi akan menampilkan pesan “Client could not connect!”. Berikut ini potongan kode programnya:

```
CString s;
GetDlgItem(IDC_IPADDRESS)->GetWindowText(s);
. . .
saClient.sin_addr.s_addr = inet_addr(s);
saClient.sin_family = AF_INET;
saClient.sin_port = htons(atoi(m_port));
if ( connect(sClient, (sockaddr*)&saClient, sizeof(saClient)) ==
      SOCKET_ERROR )
{
      if ( WSAGetLastError() != WSAEWOULDBLOCK )
      {
            MessageBox("Client could not connect!", "Client",
            MB_OK+MB_ICONERROR);
            return FALSE;
      }
}
```

Ketika koneksi telah terbentuk antara *client* dan *server* maka aplikasi *client* perlu melakukan inisialisasi struktur BITMAPINFOHEADER. Struktur ini berguna untuk menentukan hal-hal yang berhubungan dengan gambar yang akan ditampilkan oleh fungsi-fungsi DrawDib. Inisialisasi ini dilakukan dalam fungsi OnButtonconnect milik kelas CClientDlg. Berikut ini potongan kode programnya:

```
. . .
CWnd* w = (CWnd*)GetDlgItem(IDC_PREVIEW);
. . .
dc = ::GetDC(w->GetSafeHwnd());
pYUVFmt =(LPBITMAPINFOHEADER) malloc(sizeof(BITMAPINFOHEADER));
memset(pYUVFmt, 0, sizeof(BITMAPINFOHEADER));
pYUVFmt->biSize = sizeof( BITMAPINFOHEADER);
pYUVFmt->biWidth = 160;
pYUVFmt->biHeight = 120;
pYUVFmt->biPlanes = 1;
pYUVFmt->biBitCount = 24;
pYUVFmt->biCompression  = BI_RGB;
```

Member biWidth dan biHeight pada struktur BITMAPINFOHEADER menentukan lebar dan tinggi gambar yang akan ditampilkan. *Member* biBitCount menentukan resolusi gambar yang akan ditampilkan. Nilai 24 yang tercantum menyatakan gambar akan ditampilkan dengan resolusi 24-bit untuk tiap pikselnya.

Berikutnya pembahasan tentang status hak kontrol. Ketika pertama kali *client* berhasil melakukan koneksi ke *server*, aplikasi *server* mengirimkan pesan

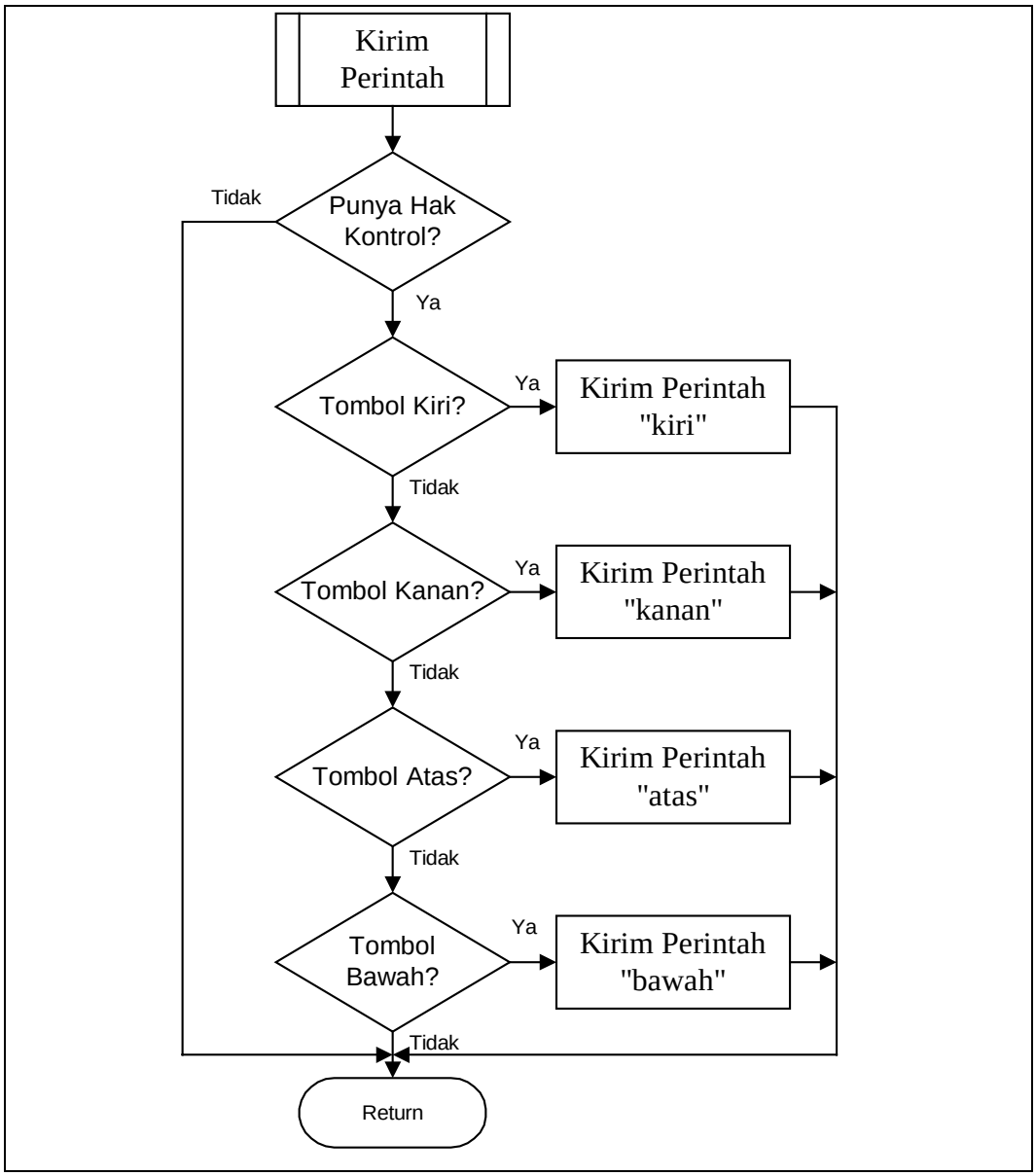
yang isinya menentukan berhak atau tidaknya aplikasi *client* melakukan kontrol terhadap kamera yang terpasang di komputer *server*. Jika *client* memperoleh hak kontrol maka aplikasi *client* akan mengaktifkan semua tombol-tombol arah yang pada waktu pertama kali aplikasi dijalankan dalam keadaan *disable*. Berikut ini potongan kode programnya:

```
result = recv(sClient,buf,9,0);
if ( result == SOCKET_ERROR )
{
      . . .
}
else
{
      if ( strcmp(buf,"control") == 0 )
      {
      GetDlgItem(IDC_BUTTONUP)->EnableWindow(TRUE);
      GetDlgItem(IDC_BUTTONRIGHT)->EnableWindow(TRUE);
      GetDlgItem(IDC_BUTTONDOWN)->EnableWindow(TRUE);
      GetDlgItem(IDC_BUTTONLEFT)->EnableWindow(TRUE);
      have_ctrl = 1;
      }
      send(sClient, s, s.GetLength(), 0);
      mode_rec = 1;
}
```

3.1.2.2. Prosedur Kirim Perintah

Prosedur kirim perintah melakukan hal-hal yang berhubungan dengan pengiriman pesan yang berisi perintah untuk menggerakkan kamera ke komputer *server*.

Diagram alur prosedur kirim perintah adalah sebagai berikut:



Gambar 3.16. Diagram Alur Prosedur Kirim Perintah

Pengiriman pesan ini menggunakan fungsi *send* milik winsock. Namun demikian, seperti yang sudah dijelaskan pada bagian sebelumnya, tidak semua *client* dapat mengirimkan pesan ke *server*, hanya *client* yang mempunyai hak kontrol saja yang dapat melakukannya. Berikut ini potongan kode program untuk mengirimkan perintah ke *server*:

```
void CClientDlg::OnButtonup()
{
	CString s;
	s.Format("ATAS");
	send(sClient, s, s.GetLength(), 0);
}
```

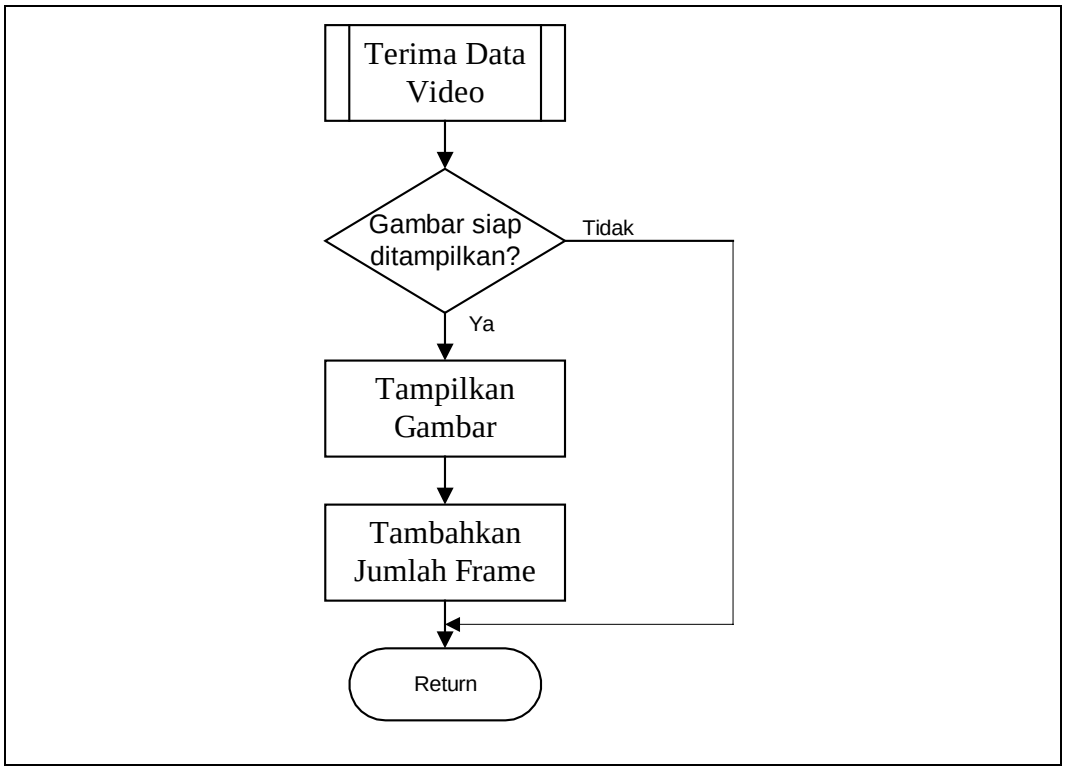
```
void CClientDlg::OnButtonright()
{
      CString s;
      s.Format("KANAN");
      send(sClient, s, s.GetLength(), 0);
}

void CClientDlg::OnButtondown()
{
      CString s;
      s.Format("BAWAH");
      send(sClient, s, s.GetLength(), 0);
}

void CClientDlg::OnButtonleft()
{
      CString s;
      s.Format("KIRI");
      send(sClient, s, s.GetLength(), 0);
}
```

Setelah *client* yang memiliki hak kontrol melakukan *disconnect*, hak kontrol tidak diberikan pada *client* yang lain. Jika ada *client* baru yang melakukan koneksi ke *server*, hak kontrol akan diberikan pada *client* baru tersebut. Jadi hak kontrol tersebut akan ditahan *server* sampai ada *client* baru yang terkoneksi.

3.1.2.3. Prosedur Terima Data Video

Prosedur terima data video melakukan hal-hal yang berhubungan dengan penerimaan data video yang dikirim *server* melalui jaringan serta menampilkan data video tersebut ke layar dalam bentuk gambar. Untuk menampilkan gambar ke layar, aplikasi *client* memanfaatkan fungsi-fungsi DrawDib.

Diagram alur prosedur ini adalah sebagai berikut:



Gambar 3.17. Diagram Alur Prosedur Terima Data Video.

Setelah komputer *client* terhubung dengan *server*, *server* menentukan pemberian hak kontrol, kemudian *client* menerima pemberitahuan *server* tersebut dan membalasnya dengan mengirimkan pemberitahuan untuk segera memulai proses *streaming* dan mengirimkan data video ke *client* secara terus-menerus sampai koneksi dengan *server* putus.

Aplikasi *client* menerima data video dari *server* dengan menggunakan fungsi *recv* milik winsock. Dalam sekali waktu, fungsi *recv* hanya menerima sebagian dari data video yang dikirimkan *server*. Jadi data video yang sedikit demi sedikit masuk ke *client* tersebut, disimpan dulu pada sebuah variabel sebelum ditampilkan secara utuh ke layar. Berikut potongan kode programnya:

```
int fps = 0;
int result;
int lake = 57600;
int offset = 0;
char buff[57600];
. . .
result = recv(sClient, buff+offset, lake, 0);

if ( result == SOCKET_ERROR )
{
      . . .
}
```

```
else
{
      offset += result;
      lake -= result;

      if (lake == 0)
      {
            . . .
            offset = 0;
            lake = 57600;
            fps++;
      }
}
```

Variabel buff merupakan *array* char dengan kapasitas 57600 *byte*. Variabel ini digunakan untuk menampung data video yang diterima dari *server*. Variabel offset digunakan untuk menunjukkan sampai bagian mana variabel buff sudah terisi data video, juga sekaligus digunakan untuk menentukan mulai dari mana data video yang datang berikutnya akan ditempatkan dalam variabel buff. Variabel lake digunakan untuk menentukan jumlah *byte* yang masih dapat ditampung oleh variabel buff. Sedangkan variabel result digunakan untuk mengetahui jumlah *byte* yang diterima variabel buff untuk sekali pemanggilan fungsi *recv*.

Ketika sejumlah *byte* berhasil diterima oleh fungsi *recv*, maka nilai variabel result akan ditambahkan ke nilai variabel offset, karena dengan adanya data yang masuk ke variabel buff, variabel offset harus berubah supaya ketika ada data baru yang masuk, data tersebut dapat dengan benar ditempatkan pada variabel buff. Kebalikan dengan variabel offset, nilai variabel lake justru dikurangi nilai variabel result, karena dengan semakin banyaknya isi variabel buff, jumlah *byte* yang dapat ditampung akan semakin sedikit.

Dengan demikian, jika variabel lake berisi 0 (nol) berarti variabel buff sudah terisi penuh dengan data video dan siap untuk ditampilkan ke layar. Setelah fungsi-fungsi DrawDib mengubah data video pada variabel buff menjadi gambar dan menampilkannya ke layar, maka nilai variabel offset dan variabel lake perlu dikembalikan ke nilai awalnya masing-masing untuk penerimaan data video pada *frame* berikutnya. Variabel fps digunakan untuk menentukan jumlah *frame* yang diterima *client* setiap satu detik. Tiap satu *frame* siap ditampilkan, variabel fps akan ditambah dengan satu.

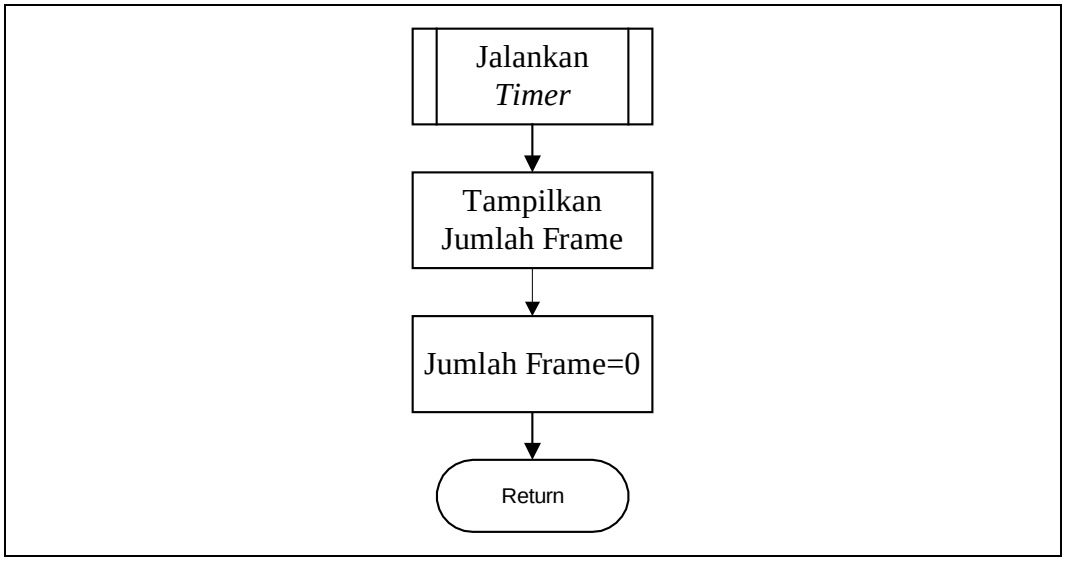
Fungsi-fungsi DrawDib akan menampilkan data video yang tersimpan pada variabel buff ke layar. Dalam hal ini, fungsi-fungsi DrawDib yang digunakan adalah fungsi DrawDibBegin dan fungsi DrawDibDraw. Berikut potongan kode programnya:

```
if (lake == 0)
{
      DrawDibBegin(dibdc,NULL,0,0, pYUVFmt,160, 120, NULL);
      DrawDibDraw(dibdc, dc, 20,20,160,120,pYUVFmt, (LPBYTE)buff,
      0,0, 160, 120, DDF_SAME_DRAW | DDF_SAME_HDC);
      . . .
}
```

3.1.2.4. Prosedur Jalankan *Timer*

Di dalam prosedur ini, dijalankan *timer* windows dan akan melakukan pemanggilan fungsi OnTimer tiap satu detik. Dalam fungsi OnTimer inilah, jumlah *frame* yang diterima *client* tiap satu detik ditampilkan. Jadi prosedur ini secara tidak langsung berhubungan dengan prosedur terima data video. Dalam prosedur terima data video dilakukan perhitungan jumlah *frame* perdetik, sedang jumlah *frame* perdetik itu akan ditampilkan pada prosedur jalankan *timer* ini. Selain itu, ditampilkan juga lama waktu aplikasi *client* dijalankan.

Berikut ini adalah diagram alur prosedur jalankan *timer*:



Gambar 3.18. Diagram Alur Prosedur Jalankan *Timer*

Pemanggilan fungsi OnTimer ini dilakukan secara otomatis setiap satu detik seperti yang sudah didefinisikan pada bagian inisialisasi *timer* dengan menggunakan fungsi SetTimer. Berikut potongan kode programnya:

```
int t = 0;
. . .
void CClientDlg::OnTimer(UINT nTimerID)
{
      CClientDlg* dlg = (CClientDlg*)AfxGetApp()->GetMainWnd();
      CString s;
      t++;
      s.Format("%d : %d fps",t,fps);
      dlg->GetDlgItem(IDC_FPS)->SetWindowText(s);
      fps = 0;
}
```

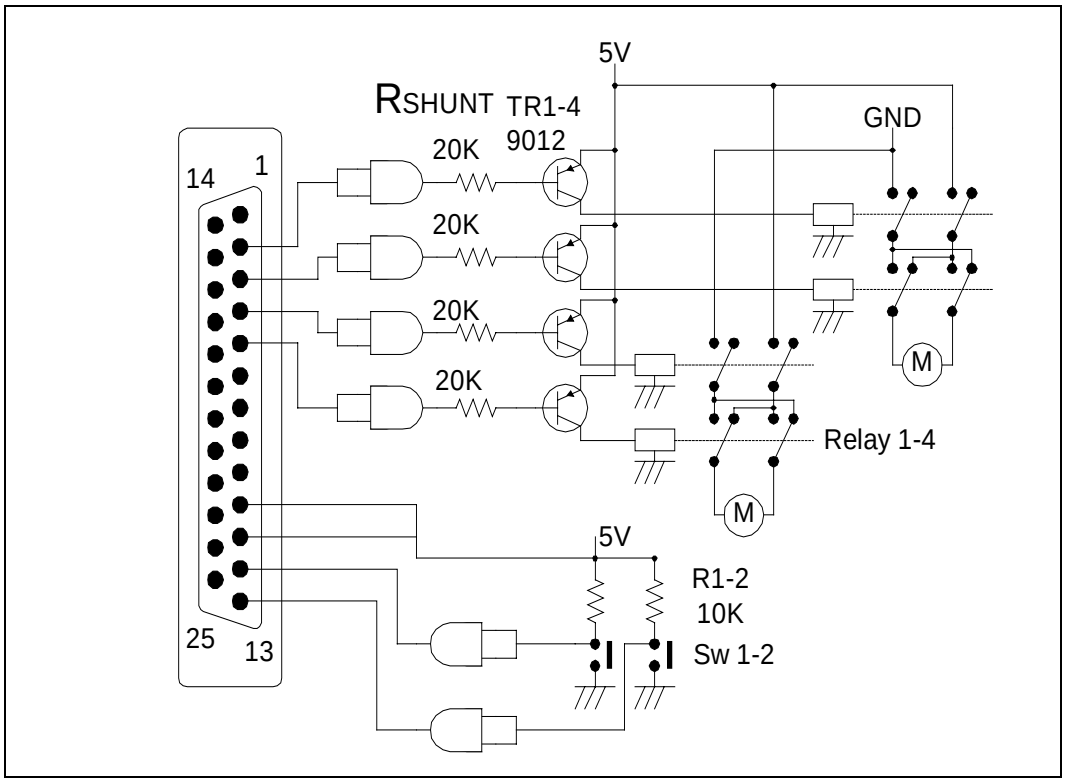
Variabel t digunakan untuk menunjukkan lama aplikasi dijalankan. Variabel s digunakan untuk menampung lama aplikasi dijalankan dan jumlah *frame* perdetik. Lama aplikasi dijalankan dan jumlah *frame* perdetik yang disimpan dalam variabel s akan ditampilkan ke kontrol *text field* dengan menggunakan fungsi SetWindowText.

3.2. Perancangan *Hardware*

Perangkat keras sistem terdiri dari rangkaian *driver* dan kerangka penyangga. Rancangan untuk rangkaian *driver* terdiri dari komponen-komponen elektronika, kabel-kabel yang dibutuhkan dan 2 buah motor DC yang terhubung dengan rangkaian. Sedangkan rancangan kerangka penyangga hanya terdiri dari menara mini yang terbuat dari bahan kayu dan triplek.

3.2.1. Rangkaian *driver*

Rangkaian *driver* terdiri dari sebuah pcb lubang, 4 buah *relay* 5V, sebuah IC TTL 74LS08, 4 buah TR 9012, 4 buah *resistor* 10K, 2 buah *limit switch*, 2buah motor DC, sebuah kabel data dengan konektor DB 25 *male* dan sebuah kabel USB yang digunakan sebagai *power supply*. Gambar 3.19 adalah adalah skema rangkaian *driver* untuk perangkat keras sistem.



Gambar 3.19. Rangkaian *Driver* Sistem

Prinsip kerjanya sederhana, pin ke-2 sampai pin ke-5 masing-masing masuk ke gerbang-gerbang AND yang kedua kaki masukannya dihubungkan. Gerbang-gerbang AND tersebut berfungsi sebagai *buffer*. *Relay* 5V, mendapat tegangan *drive* dari *transistor* 9012, jika selisih tegangan antara kaki B dan kaki E *transistor* adalah sebesar 0,7V (V_{BE} = 0,7V).

Karena *transistor* 9012 adalah *transistor* jenis PNP maka pembacaan dibalik sehingga tegangan di kaki E lebih besar dari tegangan di kaki B, ini berarti bahwa ambang tegangan transistor untuk aktif adalah sebesar $0,7V_{EB}$.

Bila tegangan pada emitor (V_E) 5V, maka tegangan yang dibutuhkan basis:

$$V_B = V_E - 0,7V$$

$$V_B = 5V - 0,7V$$

$$V_B = 4,3V$$

maka kaki basis *transistor* 9012 membutuhkan tegangan 4,3V untuk aktif dan 0V untuk *cut-off*.

Transistor 9012 mempunyai faktor penguatan (Hfe) sebesar 200. Arus yang mengalir pada *relay* (I_C) sebesar 37,5mA, dengan demikian I_B membutuhkan arus sebesar:

$$I_B = \frac{I_C}{Hfe} = \frac{37{,}5mA}{200} = 0{,}1875mA$$

maka:

$$R_{SHUNT} = \frac{V_B}{I_B} = \frac{4{,}3V}{0{,}1875mA} = 22{,}93K\Omega$$

Nilai R_{SHUNT} dibulatkan menjadi 20KΩ untuk dapat mengantisipasi beban *over drive*. Jika nilai R_{SHUNT} yang digunakan adalah 20KΩ maka nilai I_B menjadi:

$$I_B = \frac{V_B}{R_{SHUNT}} = \frac{4{,}3V}{20K\Omega} = 0{,}215mA$$

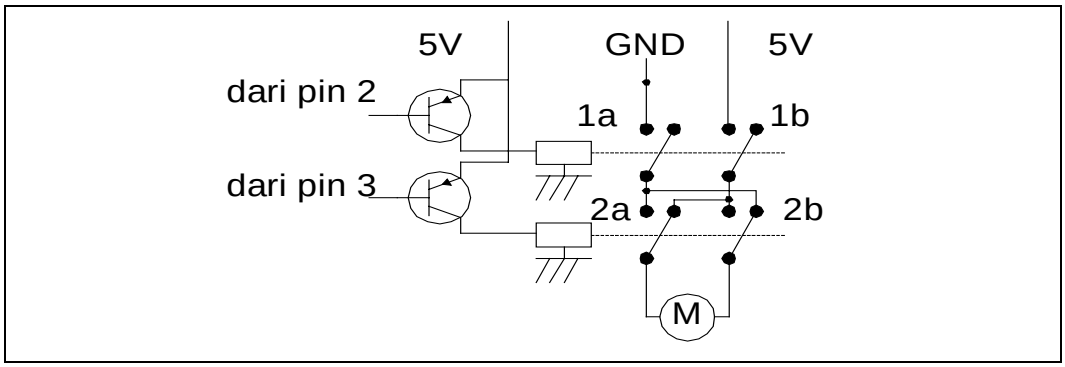
maka:

$$I_C = I_B \times Hfe$$

$$I_C = 0{,}215mA \times 200 = 43mA$$

Maka, bila *relay* membutuhkan arus lebih untuk aktif, disediakan arus sebesar 43mA, sedangkan yang dibutuhkan hanya 37,5mA.

Relay-relay yang terdapat pada rangkaian masing-masing memiliki fungsi yang berbeda. Dua buah *relay* memiliki fungsi sebagai pengatur hidup dan mati motor, sedangkan dua lainnya sebagai pengatur arah putaran motor. Untuk lebih jelasnya, lihat gambar 3.20 di bawah ini.



Gambar 3.20. Sistem Kerja *Relay*

Sebagai penjelasan, kita ambil contoh dua buah *relay* yang terhubung dengan motor vertikal (*relay* 1 sebagai *enable* dan *relay* 2 sebagai pengatur arah). Pada waktu *relay* 1 dalam keadaan tidak mendapat tegangan *drive* dari *transistor*, saklar 1a dan saklar 1b tidak terhubung dengan apa-apa. Apapun juga kondisi *relay* 2, tidak akan membuat motor berputar karena tidak ada *supply* untuk motor.

Namun jika *relay* 1 mendapat tegangan *drive* dari *transistor*, sehingga saklar 1a terhubung dengan *ground* (GND) dan saklar 1b terhubung dengan V_{CC} 5V, maka *relay* 2 baru berfungsi, yaitu menentukan ke arah mana motor akan berputar. Ada atau tidak tegangan *drive* dari *transistor* pada *relay* 2 berpengaruh pada arah putaran motor, karena tegangan *drive* pada *relay* 2 akan mengubah posisi saklar 2a dan saklar 2b, perubahan posisi kedua saklar tersebut akan menyebabkan pembalikan polaritas pada motor DC. Seperti yang kita ketahui, pembalikan polaritas pada motor DC akan membalik arah putaran motor tersebut.

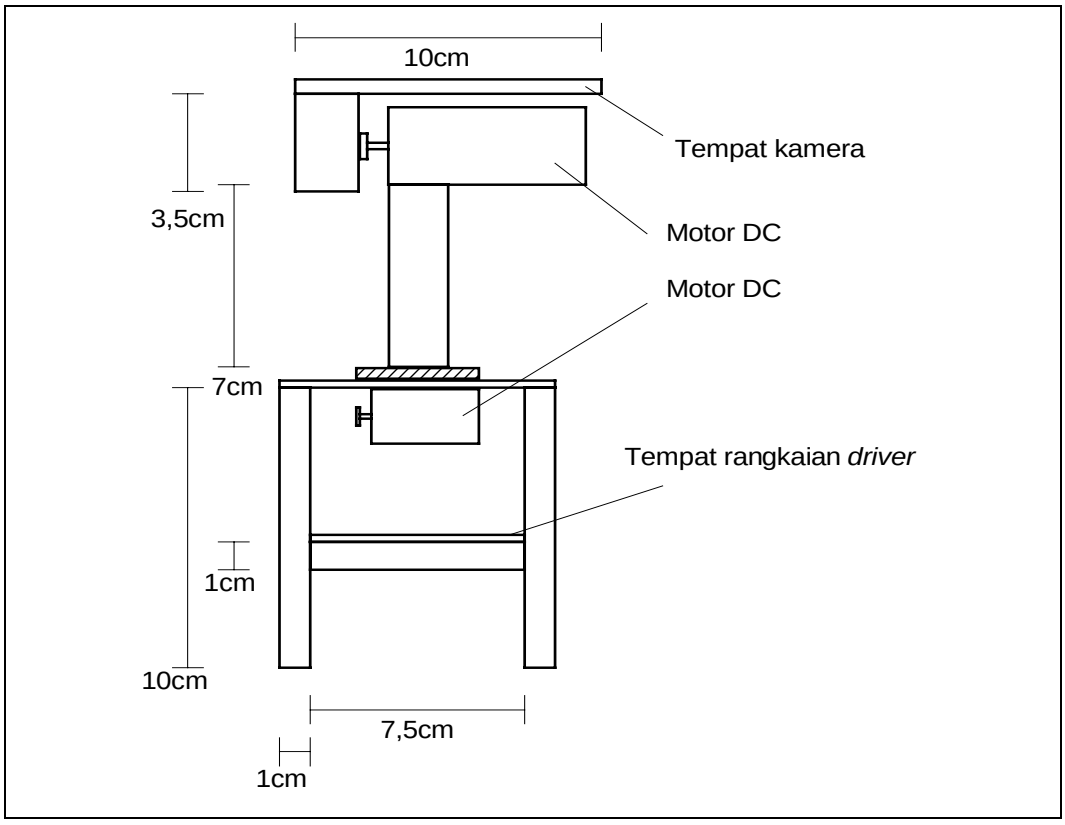
Power supply untuk rangkaian driver diambil dari USB *port* dengan memanfaatkan kabel USB. Alasan penggunaan USB *port* dan kabel USB ini untuk mendukung pemberian *supply* pada rangkaian *driver* adalah karena kepraktisannya, mudah dibawa dan murah. Benar-benar praktis karena dengan memanfaatkan kabel USB, tidak perlu membawa rangkaian *power supply* yang kemungkinan besar dan berat. Mudah dibawa karena kabel USB bentuknya kecil bahkan bisa langsung dipasang secara permanen pada rangkaian *driver*.

3.2.2. Kerangka Penyangga

Kerangka penyangga dirancang untuk menyangga kamera, motor dan rangkaian *driver*. Kerangka penyangga dibuat berdasarkan rancangan pada

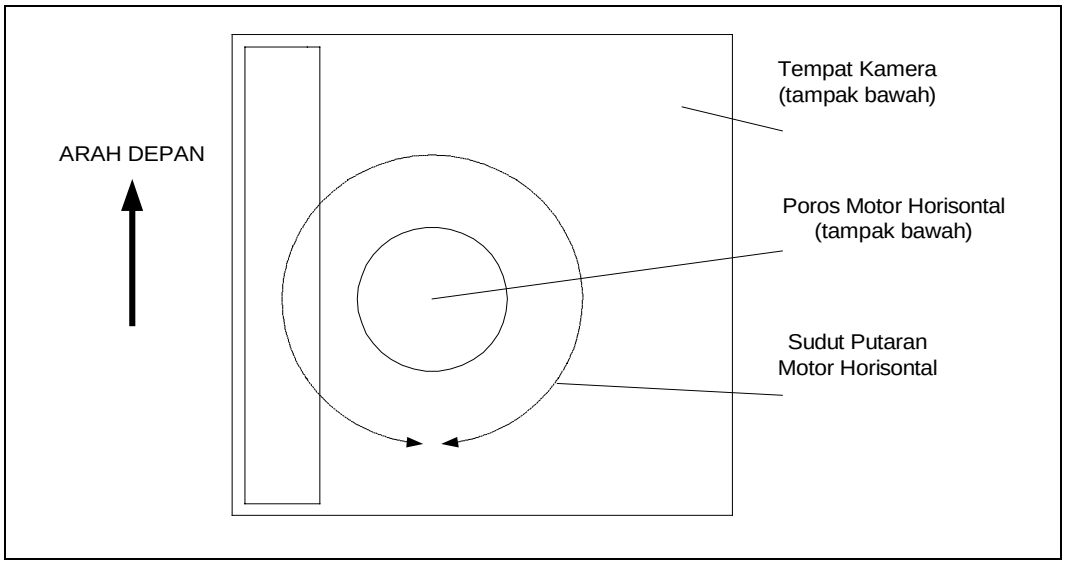
gambar 3.21 dengan bahan baku yang sebagian besar adalah kayu. Alasan pemilihan bahan tidak lain adalah: bahan mudah diperoleh, cukup kuat dengan beban yang tidak terlalu berat serta mudah dalam hal pemotongan dan pemasangannya.

Ukuran-ukuran yang ada pada hasil implementasi disain perangkat keras diupayakan sama dengan ukuran yang ada pada rancangannya. Karena faktor-faktor tertentu, ukuran yang ada pada kerangka yang sebenarnya tidak sama persis dengan yang ada pada disain.



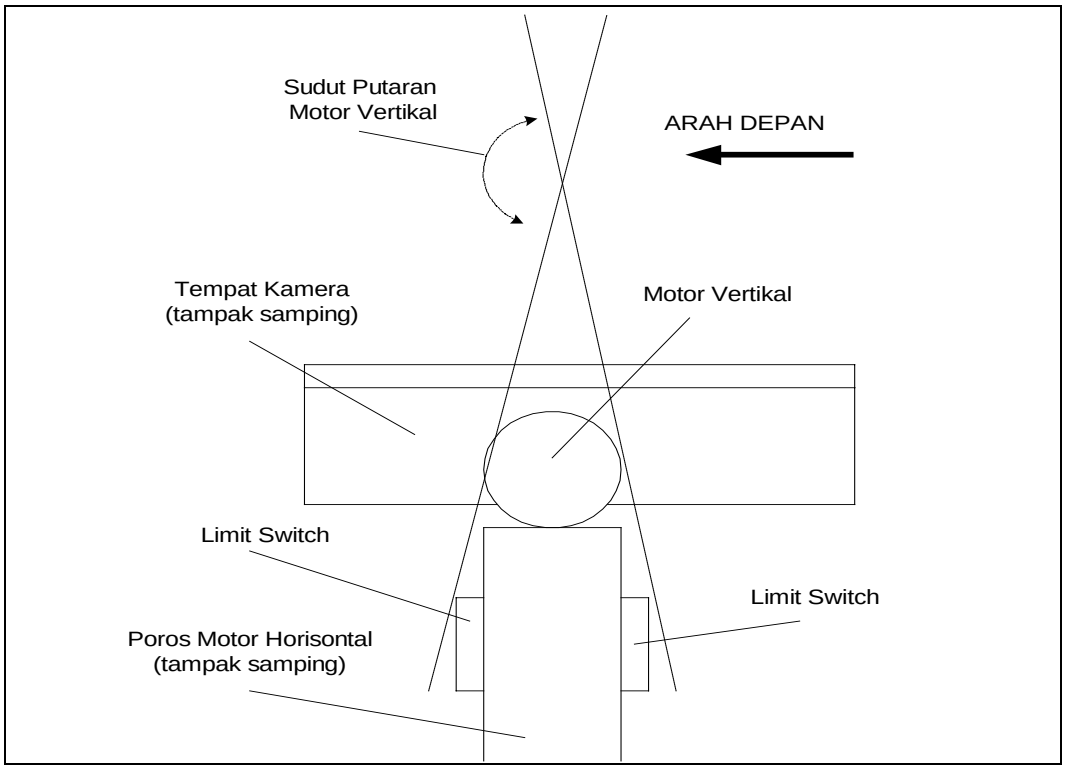
Gambar 3.21. Rancangan Kerangka Penyangga

Tempat kamera yang berada di bagian atas mempunyai keistimewaan, yaitu kamera dapat dipasang dan dilepas dengan mudah. Motor vertikal berada tepat di bawah tempat kamera tersebut, sedangkan motor horisontal berada di atas tempat rangkaian *driver*. Motor horisontal memiliki *worm gear* sehingga bila putaran sudah mencapai batasnya (pada 360°), motor sudah tidak mau berputar lagi.



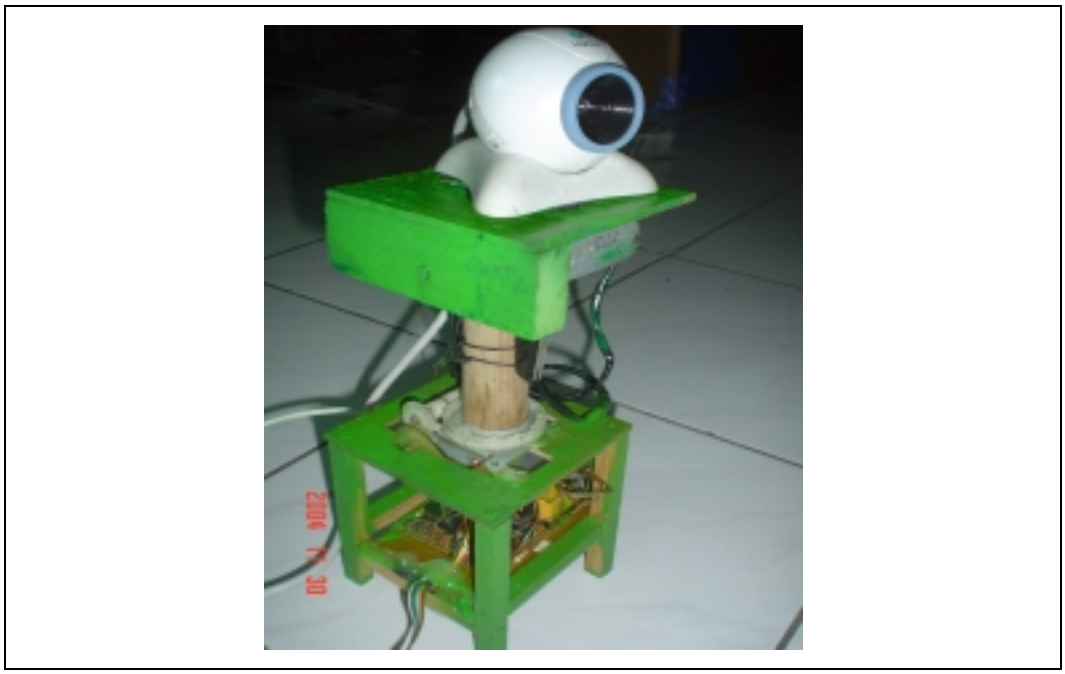
Gambar 3.22. Sudut Putar Motor Horisontal

Motor vertikal tidak mempunyai *worm gear* seperti pada motor horisontal. Putaran pada motor vertikal ini dibatasi dengan 2 buah *limit switch* yang terpasang pada leher kerangka, satu dipasang di bagian depan leher kerangka, sedang yang lain dipasang di bagian belakang leher kerangka. Pada saat tempat kamera bagian depan menyentuh *limit switch* depan, motor vertikal akan dimatikan secara *software* sehingga tidak lagi berputar kecuali arah putarannya dirubah. Sebaliknya, saat tempat kamera bagian belakang menyentuh *limit switch* belakang, motor vertikal juga akan dimatikan sampai arah putaran motor dirubah. Sudut yang terbentuk antara 2 keadaan tersebut merupakan sudut putaran yang dihasilkan motor vertikal, yaitu sebesar 165°.



Gambar 3.23. Sudut Putar Motor Vertikal

Berikut ini hasil implementasi dari rancangan perangkat keras sistem yang diambil gambarnya dari beberapa sudut pandang.



Gambar 3.24. Perangkat Keras Sistem Tampak Depan Kanan

Gambar 3.25. Perangkat Keras Sistem Tampak Depan Kiri