Video Surveillance and Monitoring
FRE Spec Document 98-1.0

Carnegie Mellon University
Sarnoff Corporation

February 19, 1998

Contents

1 Contact details

For any further information or feedback, please contact:

Dr Alan Lipton
The Robotics Institute
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA, 15213

Phone: (412) 268 1450
Fax: (412) 268 5571
Email: ajl@cs.cmu.edu
Url: http://www.cs.cmu.edu/~ ajl

2 Expected domain of FRE Participation

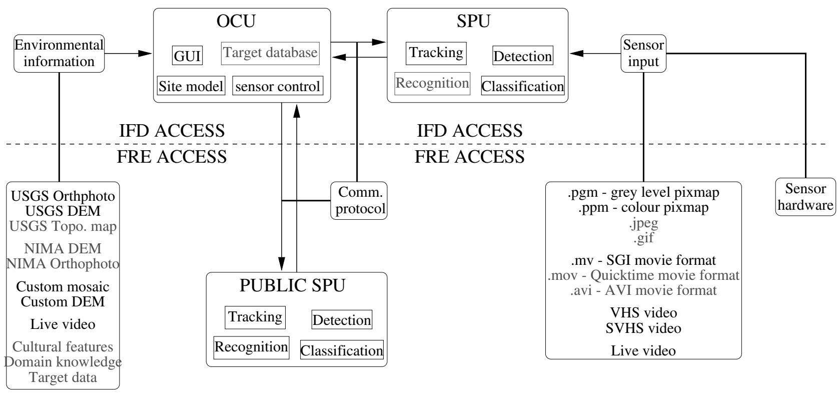

Figure 1: Proposed logical system infrastructure with tap points for FREs. Refer to the VSAM '97 Spec document for descriptions of the operator control unit (OCU) sensor processing units (SPUs)

Figure 1 shows the proposed system for 1998 with tap points for FREs (or other interested parties). The IFD will accept data from FREs (or other parties) in the form of

- Site model data
  - USGS/NIMA orthophotos, DEMS, topographic maps
  - Custom images/DEMS (in a format TBD)
  - Cultural/domain/target database entries (in a format TBD)
- Video data
  - Live video from FRE sensors

  - VHS/S-VHS video tape
  - Movie files (in QT, SGI or MPEG format)
  - Image sets (.gif, .jpeg, .ppm, .pgm)
- Target/sensor data in IFD specified protocol

Six levels of FRE participation in Demo II have been considered of which five will be supported in 1998

1. Sensors - FREs provide sensors for IFD testbed
2. Live taps - IFD provides live video taps during the demo which the FREs can use as inputs to their systems
3. Parallel system - FREs can set up their systems (cameras, processors and visualisation) in parallel with the IFD. Output can also be shown in parallel. (Assuming FREs do not absorb critical band-width or line-of-sight at the demo site).
4. FRE SPU - Using the protocol outlined in this document, FREs can provide a complete SPU to be integrated with the IFD testbed system.
5. FRE Visualisation tool - Using the protocol outlined in this document, FREs can provide a visualisation tool using data collected from IFD SPUs.
6. FRE algorithms (not supported in 1998) - FREs can provide algorithms which are integrated into IFD SPUs.

A list of approximate dates for FRE participation is given below:

Feb. 16, 1998	First publication of FRE participation spec.
Mar. 24, 1998	Comments due from FRE on Spec. Document.
Apr. 16-17, 1998	VSAM workshop III (Ft Benning) FRE spec. finalised based on IFD/FRE discussions.
May 15, 1998	FRE's inform IFD if they intend to participate in the '98 IFD demo.
Jun. 12, 1998	Demo planning workshop. FRE participation is mandatory for FREs with levels 2-5 of participation.
Aug-Sep (TBD), 1998	Various FRE participation days/weeks during which IFD & FREs will work at the demo site to install and test FRE systems.
Oct 8-9, 1998	Proposed DEMO II

3 Code organisation and distribution

An FRE participation will be set up and maintained by the IFD which will contain the following information:

- The latest version of the FRE participation spec. document.
- A list of updates to the document and the dates they were implemented
- Sample code segments containing:
  - `SPU.h` - header file containing relevant data structures for SPU and visualisation tool code
  - `SPU.cpp` - SPU packet processing object which will contain code for packing and unpacking communication packets.
  - `dataline.h` - header file for Ethernet communications.
  - `dataline.cpp` - C++ Ethernet communications object.

In addition to this web-site, an interested party mailing list will be created and maintained by the IFD to announce updates to the FRE participation spec., and allow feedback from FREs.

4 Physical connexion to IFD testbed

In the proposed implementation of the IFD testbed system for Demo II (1998), two separate data transmission systems will be implemented: one exclusively for raw or partially processed video imagery and the other for logical data traffic between SPUs and OCU components. Within the IFD testbed, video imagery will be transmitted by wireless microwave video links and standard video cables, and logical data will be transmitted around a local area network via wireless Ethernet and regular Ethernet cables in a standard Internet protocol.

4.1 Data connexion

All FREs wishing to provide an SPU or visualisation tool will be able to access IFD testbed logical data (as opposed to video imagery) over the local area network. A number of standard 10-baseT ports with RJ-45 connexion jacks will be made available to FREs from a central IFD testbed Ethernet hub.

FREs will be responsible for providing their own Ethernet cables and connectors, or other Ethernet transmission system. Space at the operator control station may be limited, so FREs should expect their equipment to be positioned up to 25 feet distant from the IFD testbed Ethernet hub.

4.2 Video connexion

All IFD testbed signals will be in analogue NTSC format and will be provided on standard BNC or RCA jacks. The minimum specification (guaranteed by the IFD) for SPU data will be:

- Modality: monochrome
- Resolution: 320×240 pixels
- Frame Update Rate: > 7 frames/second
- Preprocessing: None

However, some SPUs may provide higher quality imagery.

FREs wishing to exploit live raw, or partially processed video, from the IFD testbed will be able to gain access through standard analogue NTSC video transmission means. FREs will have access to two modes of video data: direct signals, and space-multiplexed signals.

Each IFD SPU will have a limited number of direct signal taps available for FRE participation. These will be in the form of standard RCA or BNC jacks.

In addition to these taps, there will be one space-multiplexed video output containing 4 or 8 SPU imagery streams on a single NTSC signal. The IFD cannot guarantee which SPUs' imagery will be available on this signal at any given time during the demo.

FREs are required to provide their own video cables and connectors (marked with coloured tape or otherwise for easy identification), or other video transmission system. All connectors to be standard BNC or RCA (FREs to provide appropriate adapters if necessary), and all cables to be standard 75Ohm coaxial, RG-59 video cable. Space at the operator control station may be limited, so FREs should expect their equipment to be positioned up to 25 feet distant from the IFD testbed.

4.2.1 Caveat - video quality

The IFD cannot take responsibility for video signals which are degraded due to noise in the acquisition or transmission process, or any other source. FREs are warned that where wireless video transmission is used (particularly in the case of the airborne video sensor) there is often considerable video distortion and noise. Furthermore, due to transmission limitations, some SPUs may provide post-processed video data for some or all of their operational time. A concerted effort will be made by the IFD team to ensure that raw video imagery is available as often as possible.

4.3 Remote SPUs

FREs providing remote SPUs will be responsible for their providing their own infrastructure. The issues involved are:

- Ethernet connexion. FREs will be responsible for setting up and maintaining a physical data link to the IFD OCU. This may be in the form of Ethernet cabling (provided by the individual FRE), or wireless ethernet transmission system.

FREs may provide their own self-contained wireless Ethernet transmission system or they can take advantage of the Breezecom(tm) system set in place by the IFD.

- Power supplies. FREs will also be responsible for providing their own power supplies, although power may be available at the FREs SPU site depending on placement.
- Video transmission. FREs will be responsible for providing means to transmit video imagery to the IFD OCU (if such is required). This may be in the form of cabling or wireless video transmission.

4.4 FRE sensors

FREs providing vision sensor hardware for use within IFD SPUs (participation level 1) must provide SDK libraries and documentation. Libraries must be compatible with Microsoft(tm) Visual C++ 5.0 (Developer's Studio) running under windows NT(tm).

4.4.1 Specialised video hardware/drivers

If FREs have specialised computational hardware and/or hardware drivers they must comply with the IFD SPU system specification:

- System: Intel(tm) Pentium(tm) 200+Mhz or Intel(tm) Pentium II(tm) 300+Mhz processor
- Control Bus: Standard PCI or ISA
- Operating System: Microsoft (tm) Windows(tm) NT(tm) (or Microsoft(tm) Windows 95(tm) - STRONGLY DISCOURAGED!!!)
- Code Compatibility: Any required code, linked libraries or run-time libraries (DLLs) must be compatible with Microsoft(tm) Visual C++ 5.0 (Developer's Studio).

5 Communication protocol

This section describes the communications protocol which will be used to relay messages through the IFD testbed system's local area network. Sample SPU communication code (in C++) is given in the appendix and will be made available to FREs on-line or by request.

5.1 Internet protocol

All data communications in the IFD local area network will be conducted under the standard asynchronous Internet protocol TCP/IP.

5.2 IFD Clients - the comms protocol

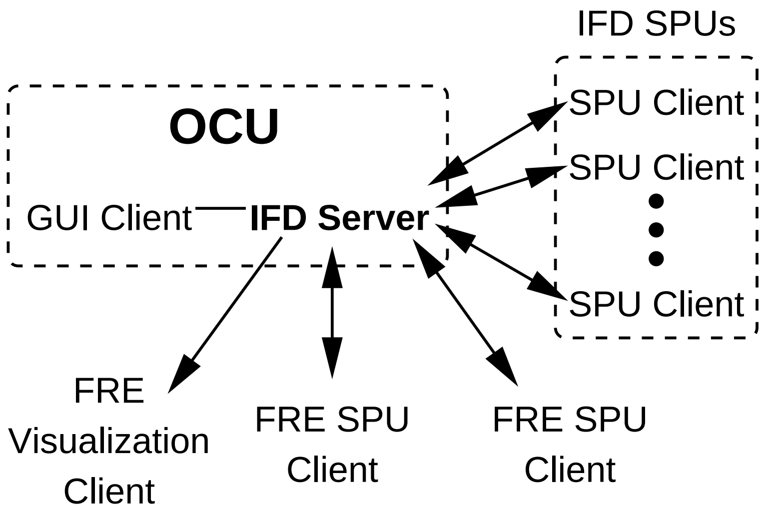

Figure 2: Client/server model of the IFD LAN

The philosophy behind the IFD testbed system is that each SPU is a relatively unintelligent entity. It is capable of image processing, active tracking, and communication with the OCU, but is unaware of broader issues such as site models and the presence of other SPUs. All of this information resides at the OCU. The reason for this choice is to minimise the amounts of redundant information in the system - which, in turn, makes data book-keeping simpler. If FRE SPUs require detailed site models or other system knowledge, it will be the responsibility of the FRE to collect and maintain this information within their individual SPU.

Figure 2 shows the connexion model of the IFD LAN. It is a simple client/server model with two types of clients supported: SPU clients, and Visualisation clients. SPU clients are actual sensor units which receive control signals from the IFD OCU and transmit target information back to the OCU. Visualisation clients are passive and receive target and sensor information from the IFD OCU.

The communication protocol employed on the IFD LAN is an asynchronous trasmit without acknowledge protocol. The notion is that SPUs continuously and asynchronously transmit their "latest" status data and the OCU prioritises incoming information, processes it, and passes it (asynchronously) on to the GUI (and other visualisation clients). Any SPU control command that is issued by the IFD GUI, or any command generated within the OCU, is asynchronously transmitted to the relevant SPU. If the SPU does not comply with the command, it is re-sent. Compliance is not determined explicitly, but is inferred from the packets which are returned from the SPU.

5.3 SPU-OCU communication

5.3.1 Communication packets

There are two types of packets which travel between the OCU and SPUs: A sensor/target status packet which is sent to the OCU by an SPU, and a sensor control packet which is sent to an SPU by the OCU.

A sensor/target status packet contains the following information[1]:

- Sensor packet header information
- Sensor position and pose in absolute coordinate system
- Internal sensor calibration parameters
- Target information

A sensor control packet contains the following information:

- OCU packet header information
- Camera control parameters
- Target information

5.3.2 Target specification

Ideally, specific targets will have unique identifiers. However, there will be situations in which individual SPUs will not know (or be unable to confirm) the presence of a specific target. When situation arises, the SPU will be responsible for supplying an identifier to the target, until it, or the IFD OCU can correct the identification. The IFD OCU will maintain records of which target identifiers have been used by which SPUs.

5.3.3 SPU commands

The current commands which could be issued to an SPU via a sensor control packet are shown in table 1

[1]see section 6 and appendices for complete packet descriptions and data structures

Command	Meaning	Arguments
0	NULL	CONTROL: NULL TARGET: NULL
1	MOVE	CONTROL: New position TARGET: NULL
2	SEARCH	CONTROL: New position TARGET: Target info

Table 1: Commands which are supported by the IFD testbed system. Column 1 is the command ID, column 2 is the command name, and column three is the type of data which the SPU can expect for its control and target information respectively.

5.3.4 Timing considerations

While the communications protocol will allow SPUs to transmit information at any rate they like, it is **HIGHLY RECOMMENDED** that the following two measures are observed to prevent bottlenecks on the LAN:

1. When there are no changes in SPU status, no packets are sent to the OCU
2. Packets are separated by a time delay of at least 250mS.

A sample communications algorithms is shown:

```
DO {
    IF (not_connected)  CONTINUE
    IF (NEW_CONTROL_PACKET()) do_control()
    IF (status == old_status)  CONTINUE
    IF (time_since_last_packet < 250mS)  CONTINUE
   SEND_PACKET()
}  UNTIL (the_cows_come_home)
```

5.4 OCU-visualisation communication

Visualisation clients will receive data from the OCU whenever new SPU data is processed. No guarantees can be made about the timing or data rate of this information. It is the FREs responsibility to deal with the OCU data output. Data from the OCU will be in the following format:

- Header information
- Sensor position/pose information - in LVCS
- Target information - in LVCS

5.5 Coordinate systems

Two coordinate systems are employed in the IFD testbed system: an absolute GPS coordinate system, and a Local Vertical Coordinate System (LVCS). All SPU data should be in the absolute geodetic coordinate system with target information being presented in image coordinates. All information within the OCU (and presented to visualisation clients) will be in LVCS to prevent computational intensity. For a detailed description of these coordinate systems and sample conversion code, see the appendix.

5.6 Connecting to the IFD server

Each FRE SPU or visualisation client will be given a unique identifier which will be issued by the IFD at the request of an FRE. These must be contained in every packet sent to the OCU server. Until a packet containing this identifier is sent to the OCU server, it will not know which client is which, so as soon as an FRE client connects to the IFD LAN, it must send an identifier packet. In the case of an FRE SPU, this packet should contain the SPU's current status. However, in the case of a visualisation client, a special ID packet has been devised (see section 6) which is to be sent every time the client connects to the network.

If, for some reason, the communication channel between the OCU server and a particular client fails, the OCU server will disconnect that client instantly. It is the responsibility of the client to attempt reconnexion with the OCU server.

6 Data structures

This section describes internal data structures which underly the data communication packets as well as the actual packet structures.

6.1 SPU - OCU packet data structures

6.2 Time

All SPUs within the IFD system must be synchronised to GPS time. It will be the responsibility of the FRE to ensure that timing is maintained on their own SPU. If the OCU receives packets which are not synchronised, it will assign "best guess" GPS timecodes to them.

When the OCU sends commands to SPUs and data packets to visualisation clients, they will contain the GPS time at which the packet is sent. This time field may be used by the FRE SPUs to confirm temporal synchronisation.

6.2.1 Supporting data structures

```
/* (SGI) Definitions of 4 byte and 2 byte ints */
typedef int   __int32;
typedef short __int16;

/* Definition of a point in SPU space */
typedef struct _point {
    double      pan,            /* sensor pan in degrees */
                tilt;           /* sensor tilt in degrees */
    double      range;          /* if 3D coords are required */
} POINT;

/* Definition of a trajectory event */
typedef struct _event {
    POINT       position;       /* target position */
    __int32     time;           /* target time */
} EVENT;

/* Definition of an "image". Data other than imagery can be contained herein*/
typedef struct _image {
    __int16     type;           /* type of data - not necessarily imagery */
    __int16     x_size, y_size; /* extent of image */
    __int16     packing;        /* data packing */
    char        *data;          /* image data */
} IMAGE;

/* Target definition within SPU */
typedef struct spu_target {
    __int16     ID;             /* Unique target ID */
    __int16     type;           /* target classification */
    __int16     top_left,       /* Target bounding box (Pixels) */
                top_right,
                bottom_left,
                bottom_right;
    __int16     traj_length;    /* length of target trajectory */
    EVENT       *trajectory;    /* past trajectory of target */
    IMAGE       image;          /* image data for target */
} SPU_TARGET;
```

Selected paramters are described in more detail below.

Parameter	Meaning	Values
IMAGE.type	descriptor of data, imagery or otherwise	0 - image
IMAGE.packing	pixel packing type	0 - 8-bit luminance 1 - 24-bit RGB 2 - 24-bit BGR 3 - 32-bit RGBA 4 - 32-bit ABGR
SPU_TARGET.type	target classification	0 - Human 100 - Vehicle

6.2.2 SPU → OCU packet data structure

```
/* SPU data structure - (to be sent to OCU) */
typedef struct spu {
    __int16     ID;             /* SPU ID */
    __int16     Type;           /* Sensor type */
    __int16     Status;         /* Current SPU status */
    __int32     Time;           /* SPU packet timecode. Units: mS */
/* Sensor GPS position (converted to LVCS in OCU) */
    double      Lat,            /* Units: degrees */
                Lon,            /* Units: degrees */
                Alt;            /* Units: metres */
/* Sensor attitude */
    double      Pitch,          /* Units: degrees */
                Yaw,            /* Units: degrees */
                Roll;           /* Units: degrees */
/* Current image centre */
    POINT       Centre;
/* Current sensor field of view */
    double      H_fov,          /* Hor. FOV. Units: degrees */
                V_fov;          /* Ver. FOV. Units: degrees */
    __int16     X_size,         /* Image size. Units: pixels */
                Y_size;
/* Sensor internal parameters */
    double      Zoom;
    double      Iris;
    double      Focus;
    double      Kappa;          /* Radial distortion */
/* Sensor limits */
    POINT       Top_left,       /* limits of the sensor's FOV */
                Bottom_right;
/* Current target information */
    char        Number_of_targets;
    SPU_TARGET  *Target;        /* Targets tracked */
```

```
} SPU;
```

Selected paramters are described in more detail below.

Parameter	Meaning	Values
SPI.ID	individual SPU descriptor provided by IFD	
SPI.Status	Current status of SPU	0 - Idle 1 - Tracking
SPU.Type	Sensor type	0 - Monochrome daylight 1 - Colour daylight 2 - IR - black hot 3 - IR - white hot 10 - Omnicam 20 - SVM

Figure 3 shows the definitions of Pitch, Yaw and Roll for both the LVCS and geodetic coordinate systems.

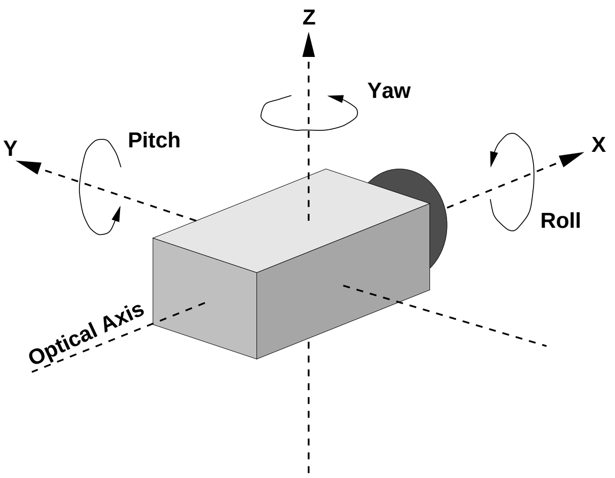

Figure 3: Definitions of Pitch, Yaw, Roll in both LVCS and geodetic coordinate sytems.

6.2.3 OCU → SPU command data structure

```
/* Packet from OCU - SPU */
typedef struct ocu {
    __int16     ID;             /* SPU ID */
    __int32     Time;           /* GPS time of command */
    __int16     Command;        /* Command for SPU */
    double      Pan, Tilt;
/* Current target information */
    char        Number_of_targets;
```

```
    SPU_TARGET *Target;          /* Targets tracked */
} OCU;
```

For a list of command definitions, see table 1.

6.3 OCU - visualisation data structures

6.3.1 Supporting data structures

```
/* Definition of a 3D point in LVCS */
typedef struct _3dpoint {
    double      x,             /* LVCS x-coord (metres) */
                y,             /* LVCS y-coord (metres) */
                z,             /* LVCS z-coord (metres) */
} _3DPOINT;

/* definition of an 3D trajectory event */
typedef struct _3devent {
    _3DPOINT    position;      /* target position */
    __int32     time;          /* time of position */
} _3DEVENT;

/* Target definition from OCU */
typedef struct _target {
    __int16     ID;              /* Target ID */
    __int16     type;            /* target classification */
    _3DPOINT    position;        /* Target bounding box (Pixels) */
    __int16     traj_length;     /* length of target trajectory */
    _3DEVENT    *trajectory;     /* past trajectory of target */
    IMAGE       image;           /* target's current image data */
} TARGET;
```

Definitions of TARGET.ID and TARGET.type are as above.

6.3.2 OCU → visualisation data structure

```
/* OCU representation of an SPU packet */
typedef struct ocu_spu {
    __int16     ID;              /* SPU ID */
    __int16     Type;            /* Sensor type */
    __int16     Status;          /* Current SPU status */
    __int32     Time;            /* SPU packet timecode. Units: mS */
/* Sensor GPS position (converted to LVCS in OCU) */
```

```
    _3DPOINT    Position,        /* Sensor position */
/* Sensor attitude */
    double      Pitch,           /* Units: degrees */
                Yaw,             /* Units: degrees */
                Roll;            /* Units: degrees */
/* Current image centre (sensor pan/tilt) */
    POINT       Centre;
/* Current sensor field of view */
    double      H_fov,           /* Hor. FOV. Units: degrees */
                V_fov;           /* Ver. FOV. Units: degrees */
/* Sensor internal parameters */
    double      Zoom;
    double      Iris;
    double      Focus;
/* Sensor footprint */
    _3DPOINT    Top_left,        /* Footprint coords in LVCS */
                Top_right,
                Bottom_left,
                Bottom_right;
/* Current target information */
    char        Number_of_targets;
    TARGET      *Target;          /* Targets tracked */
} OCU_SPU;
```

6.3.3 Special visualisation → OCU ID data structure

```
/* Special Visualization ID packet */
typedef struct _visualization {
    __int16     ID;              /* Client ID */
} VISUALIZE;
```

Definitions of OCU_SPU.ID, OCU_SPU.Status, and OCU_SPU.Type are as above.

6.4 Data packing

All of these fields are packed into data packets. There are six basic data types which must be encoded into the packet structure: 1 byte int (char), 2 byte int (__int16), 4 byte int (__int32), double (measuring angles in degrees), double (measuring altitude in metres), and double(measuring miscellaneous parameters). All data types are packed LSB to MSB (to avoid any "endian" issues). All doubles are normalised into 4 byte ints and then packed as such. Furthermore, all items measured in degrees are first converted to the range $(-180.0°, 180.0°]$. The procedures for packing values into packet arrays for these six data are:

1 byte int:

```
packet[index] = value;
```

2 byte int:

```
packet[index] = LSB(value);
packet[index+1] = MSB(value);
```

4 byte int:

```
packet[index] = LSB(value);
packet[index+1] = 2SB(value);
packet[index+2] = 3SB(value);
packet[index+3] = MSB(value);
```

where `2SB` is the second significant byte and `3SB` is the third significant byte.

double (altitude):

```
(__int32)value = (value/MAX_ALTITUDE) * MAX_INT32;
packet[index] = LSB(value);
packet[index+1] = 2SB(value);
packet[index+2] = 3SB(value);
packet[index+3] = MSB(value);
```

where `MAX_ALTITUDE = 9000.0` and `MAX_INT32 = 2**31 - 1`

double (misc):

```
(__int32)value = (value/MAX_VALUE) * MAX_INT32;
packet[index] = LSB(value);
packet[index+1] = 2SB(value);
packet[index+2] = 3SB(value);
packet[index+3] = MSB(value);
```

where `MAX_ALTITUDE` is a normalisation constant which ensures that `0 < value < 1` and `MAX_INT32 = 2**31 - 1`

double (degrees):

```
(__int32)value = (value/MAX_ANGLE) * MAX_INT32;
packet[index] = LSB(value);
packet[index+1] = 2SB(value);
packet[index+2] = 3SB(value);
packet[index+3] = MSB(value);
```

where `MAX_ANGLE = 180.0` and `MAX_INT32 = 2**31 - 1`

- END OF DOCUMENT -

A Packet descriptions

OCU → SPU Command Packet	
Item	Number of bytes
Number of bytes (in packet)	2
ID	2
Time	4
Command	2
Pan, Tilt	4, 4
Number_of_targets	1
Target 1: ID	2
Target 1: type	2
Target 1: top_left, top_right, bottom_left, bottom_right	2, 2, 2, 2
Target 1: traj_length	2
Target 1: trajectory 1: position.pan, position.tilt, position.range	4, 4, 4
Target 1: trajectory 1: time	4
⋮	
Target 1: trajectory k: position.pan, position.tilt, position.range	4, 4, 4
Target 1: trajectory k: time	4
Target 1: image.type	2
Target 1: image.x_size, image.y_size	2, 2
Target 1: image.packing	2
Target 1: image.data	x_size × y_size
⋮	
Target N: ID	2
Target N: type	2
Target N: top_left, top_right, bottom_left, bottom_right	2, 2, 2, 2
Target N: traj_length	2
Target N: trajectory 1: position.pan, position.tilt, position.range	4, 4, 4
Target N: trajectory 1: time	4
⋮	
Target N: trajectory k: position.pan, position.tilt, position.range	4, 4, 4
Target N: trajectory k: time	4
Target N: image.type	2
Target N: image.x_size, image.y_size	2, 2
Target N: image.packing	2
Target N: image.data	x_size × y_size

Visualisation → OCU ID Packet	
Item	Number of bytes
Number of bytes (in packet)	2
ID	2

SPU $\rightarrow$ OCU Packet	
Item	Number of bytes
Number of bytes (in packet)	2
ID	2
Type	2
Status	2
Time	4
Lat, Long, Alt	4, 4, 4
Pitch, Yaw, Roll	4, 4, 4
Centre.pan, Centre.tilt, Centre.range	4, 4, 4
H_fov, V_fov	4, 4
X_size, Y_size	2, 2
Zoom, Iris, Focus, Kappa	4, 4, 4, 4
Top_left.pan, Top_left.tilt, Top_left.range	4, 4, 4
Bottom_right.pan, Bottom_right.tilt, Bottom_right.range	4, 4, 4
Number_of_targets	1
Target 1: ID	2
Target 1: type	2
Target 1: top_left, top_right, bottom_left, bottom_right	2, 2, 2, 2
Target 1: traj_length	2
Target 1: trajectory 1: position.pan, position.tilt, position.range	4, 4, 4
Target 1: trajectory 1: time	4
⋮	
Target 1: trajectory k: position.pan, position.tilt, position.range	4, 4, 4
Target 1: trajectory k: time	4
Target 1: image.type	2
Target 1: image.x_size, image.y_size	2, 2
Target 1: image.packing	2
Target 1: image.data	x_size $\times$ y_size
⋮	
Target N: ID	2
Target N: type	2
Target N: top_left, top_right, bottom_left, bottom_right	2, 2, 2, 2
Target N: traj_length	2
Target N: trajectory 1: position.pan, position.tilt, position.range	4, 4, 4
Target N: trajectory 1: time	4
⋮	
Target N: trajectory k: position.pan, position.tilt, position.range	4, 4, 4
Target N: trajectory k: time	4
Target N: image.type	2
Target N: image.x_size, image.y_size	2, 2
Target N: image.packing	2
Target N: image.data	x_size $\times$ y_size

OCU → Visualisation Packet	
Item	Number of bytes
Number of bytes (in packet)	2
ID	2
Type	2
Status	2
Time	4
Position.x, Position.y, Position.z	4, 4, 4
Pitch, Yaw, Roll	4, 4, 4
Centre.pan, Centre.tilt, Centre.range	4, 4, 4
H_fov, V_fov	4, 4
X_size, Y_size	2, 2
Zoom, Iris, Focus	4, 4, 4
Top_left.x, Top_left.y, Top_left.z	4, 4, 4
Top_right.x, Top_right.y, Top_right.z	4, 4, 4
Bottom_left.x, Bottom_left.y, Bottom_left.z	4, 4, 4
Bottom_right.x, Bottom_right.y, Bottom_right.z	4, 4, 4
Number_of_targets	1
Target 1: ID	2
Target 1: type	2
Target 1: position.x, position.y, position.z	4, 4, 4
Target 1: traj_length	2
Target 1: trajectory 1: position.x, position.y, position.z	4, 4, 4
Target 1: trajectory 1: time	4
⋮	
Target 1: trajectory k: position.x, position.y, position.z	4, 4, 4
Target 1: trajectory k: time	4
Target 1: image.type	2
Target 1: image.x_size, image.y_size	2, 2
Target 1: image.packing	2
Target 1: image.data	x_size × y_size
⋮	
Target N: ID	2
Target N: type	2
Target N: position.x, position.y, position.z	4, 4, 4
Target N: traj_length	2
Target N: trajectory 1: position.x, position.y, position.z	4, 4, 4
Target N: trajectory 1: time	4
⋮	
Target N: trajectory k: position.x, position.y, position.z	4, 4, 4
Target N: trajectory k: time	4
Target N: image.type	2
Target N: image.x_size, image.y_size	2, 2
Target N: image.packing	2
Target N: image.data	x_size × y_size

B Coordinate systems

B.1 3D site coordinate systems

The VSAM IFD testbed supports two 3D site coordinate systems: geodetic GPS coordinates (latitude,longitude,elevation) and Local Vertical Coordinate System (x,y,z) coordinates. Straightforward routines for conversion between GPS and Local Vertical Coordinate System (LVCS) coordinates are available (see Appendix A).

The greatest benefit of GPS is that it is a standard geospatial coordinate system, that is, an unambiguously defined coordinate system that is globally applicable. Unfortunately, even "simple" computations, such as the distance between two points, become painful as a function of (lat,lon,elev). This problem is addressed by the LVCS, which is a Cartesian (x,y,z) coordinate system in which geometric computations are straightforward. This makes it an ideal internal representation for a VSAM system that only has to process information local to the site. The drawback to LVCS, however, is that it is defined locally with respect to some arbitrary origin point, and it is possible that two SPUs from different groups might use different LVCS coordinate systems. We forsee that communication between such different, unrelated platforms (for example, the OCU calling in an airborne SPU to look at a location in the site) should use GPS coordinates to provide an unambiguous specification of point locations. However, local geometric processing at the OCU and at individual SPUs will probably occur within an LVCS.

GPS coordinates
GPS coordinates are defined with respect to the WGS84 ellipsoid, and are represented by three values: latitude, longitude, and height in meters above the ellipsoid. Latitude and longitude are represented either as a single, floating point number denoting degrees, or as three separate numbers representing integer degrees, integer minutes, and floating point seconds. Latitude ranges from -90.0 to +90.0 degrees, with positive numbers representing latitudes North of the equator, and negative numbers denoting Southern latitudes. Longitude ranges from -180.0 to 180.0, with negative numbers meaning West of the Central Meridian), and positive numbers meaning East. The alternative degrees, minutes, second notation is supported because many instrument readouts give results in that format.

LVCS coordinates
An LVCS is a Cartesian coordinate system with its origin located somewhere at or near the site of interest. Point coordinates are represented as x,y,z distances, in meters, from the origin, where the coordinate system is oriented so that the positive X axis points east, positive Y points north, and positive Z points up (parallel to gravity). All that is needed to completely specify the LVCS is the 3D geodetic (GPS) coordinate of the origin point (see Appendix A).

B.2 Sensor coordinate system

Each SPU coordinate system specifies how a point in the image maps to a 3D viewing ray in the world. This information supports VSAM processing such as multi-sensor handoff, target geopositioning, and visibility analysis.

The geometric properties of each SPU sensor are specified in the standard way via external and internal orientation parameters. The pose of the sensor (external orientation) is specified as a 3D location, in either GPS or LVCS coordinates, and a 3D orientation (roll,pitch,yaw angles). If the SPU is mounted on a pan-tilt head, the pan and tilt angles are also included as part of the sensor pose. Internal orientation includes parameters specific to each sensor, including the focal length, field of view, optic center, and lens distortion model. Internal parameters are calibrated in an off-line procedure. External (pose) parameters are calibrated in the field. To summarize, each SPU sensor coordinate system specification consists of the following items:

```
EXTERNAL ORIENTATION
  sensor 3D location in either GPS or LVCS
  sensor 3D orientation in LVCS
    platform orientation (roll,pitch,yaw)
    pan-tilt angle
INTERNAL ORIENTATION
  horizontal and vertical fields of view,
  focal length (1997 demo system used fixed focal length lenses)
  image center (1997 demo system used numeric center of image)
  radial distortion (1997 demo system assumed zero distortion)
```

C OCU site model

The OCU site model contains VSAM-relevant information about the area being monitored. This includes both geometric and photometric information about the scene, represented using a combination of image and symbolic data. Site model representations have intentionally been kept as as simple as possible, with an eye towards efficiently supporting some specific VSAM capabilities. 1) The site model must primarily support OCU workstation graphics that allow the operator to visualize the whole site and quickly comprehend geometric relationships between sensors, targets, and scene features, 2) The site model provides a geometric context for VSAM. For example, we might directly task a sensor to monitor the door of a building for people coming in and out, or specify that vehicles should appear on roads. 3) A 3D site model supports visibility analysis (predicting what what portions of the scene are visible from what sensors(so that sensors can be efficiently tasked. 4) An accurate 3D site model supports target geopositioning via intersection of viewing rays with the terrain.

C.1 Orthophoto

An orthophoto is a nadir (downlooking) image of the site as it would look under orthographic projection. The result is an image where scene features appear in their correct horizontal positions. Note that this is not necessarily what you would see if you simply took a nadir photo from an aircraft; in such images the apparent horizontal locations of scene features are displaced due to perspective effects and differences in terrain elevation. An orthophoto typically has undergone a warping process to remove these effects.

In the OCU, an orthophoto is a color or monochrome image, aligned with the horizontal axes of the underlying site LVCS. That is, image rows are aligned parallel to the LVCS X axis, and image columns are parallel to the Y axis. The position and scale of the image must be defined by specifying the image position of the LVCS origin, and the ground sample distance (horizontal resolution) of a pixel. For the 1997 VSAM IFD demo, the OCU GUI displayed a PGM-format monochrome orthophoto with 1 meter pixel spacing.

C.2 DEM

A Digital Elevation Map (DEM) is an image whose pixel values denote scene elevations. In the OCU, DEMS are aligned with the underlying site LVCS, with image rows parallel to the X axis, and image columns parallel to the Y axis. Furthermore, elevations values are LVCS Z values. This implies that the image format must support floating point pixel values. In 1997, this was achieved by using a modified PGM format where each consecutive group of 4 unsigned char pixels was reinterpreted as a signed floating point number. As with orthophotos, the image position of the LVCS origin and the horizontal resolution must be specified. It is not required that the DEM have the same horizontal offset and scale as the orthophoto.

C.3 Scene features

A symbolic site model contains representations of important semantic features such as roads, buildings, trees. This type of model was not supported in the 1997 demo system. This year, digitized USGS topo maps are being used to obtain bitmap classifications of an orthoview, by labeling all road pixels with a particular value, buildings with another, foliage areas yet another, and so on). This is a lowest common denominator representation, but avoids problems in trying to choose among many possible symbolic representations.

Buildings probably should be represented as symbolic, volumetric models, with intensity data mapped onto them. It would also be a good idea to have a symbolic representation of where the doors are, so that focus of attention regions for building entry and exit can be defined. Note, any symbolic modeling representation we choose will be selected with an eye towards ease of use on an SGI. Therefore, we will probably be favoring Inventor-style symbolic models. Note that the Inventor language coincides with the VRML 1.0 language.

C.4 Examples of input site model data

The following is a list of input site model data sources that we either used in the 1997 testbed demo system, or plan to use in 1998. Along with each source is a description of the process involved in transforming that data into a format directly supported by the OCU.

C.4.1 USGS products

The United States Geological Survey (USGS) sells several digital mapping products that can be used to create an initial site model. Typically these are referenced to an inconvenient coordinate system (such as Universal Transverse Mercator coordinates in a North American Datum), and thus significant effort was extended to convert them into lat-lon-elev coordinates with respect to WGS84, so called "GPS coordinates." Some of the USGS products available are:

```
1. Digital Orthophoto Quarter Quad (DOQQ)
2. Digital Elevation Map (DEMS)
3. Digital Topographic Map (DRG) (not used in 1997)
4. Digital Line Graph (DLG) (not used in 1997)
```

Many of these can be ordered directly from the USGS EROS Data Center web site, located at URL `http://edcwww.cr.usgs.gov/`.

C.4.2 Mosaics

This year, we demonstrated construction of mosaics using Sarnoff's VideoBrush software. Mosaic construction takes a video sequence of images and stiches them together to create an extended scene representation. The result is represented in a viewer-centered 2D coordinate system. We also demonstrated coarse registration of this 2D view with an orthophoto. This registration was performed by simple projective warping using tie points selected by hand. Some preliminary work on automated fine registration was performed - this will be a topic of further research in 1998.

C.4.3 Custom DEMS

The Robotics Institute autonomous helicopter group has mounted a high precision laser range finder onto a remote-control Yamaha helicopter. This was used to create a detailed DEM of the Bushy Run site. Raw radar returns were collected with respect to known helicopter position and orientation (using onboard altimetry data) to form a cloud of points representing returns from surfaces in the scene. These points were converted into a DEM by projecting into LVCS horizontal-coordinate bins, and computing the mean and standard deviation of height values in each bin. Note: because the VSAM team worked closely with the helicopter group, the LVCS used coincided with the VSAM demo LVCS, making integration into the OCU very easy.

C.4.4 Live video

Input of live video for display on the OCU was performed for all SPUs used in the 1997 demo, even for the air sensor. This data was mapped into small panes at the top of the OCU display screen. More sophisticated use of incoming video would be to map live video onto the 3D site model, for instance by projecting is onto a quadrilateral footprint, or even onto a 3D triangulated DEM. This would allow the user to view video data in its correct spatial relationship with the 3D scene. This was not attempted in 1997, and may not even be attempted this year.

C.4.5 VRML models

A VRML model of one of the buildings and surrounding terrain at Bushy Run was created by a local CMU spin-off company call K^2T using their factorization modeling package. The models were created from a combination of aerial and ground-based video data. The resulting models were much too large to be easily viewed and manipulated by the IFD testbed system, so this information was not unused during the demo.

D Geographic Transforms in C

The following C program fragments are provided to illustrate how to transform between geodetic GPS coordinates and Cartesian LVCS coordinates. First, some constants and helper macros are defined.

```
/**********************************************************
 * GEOTRANSFORMS.C                                        *
 * Routines for converting between coordinate systems     *
 * mainly used for transforming between GPS coords and LVCS  *
 * Author: Bob Collins    Created: Sat Oct 25, 1997       *
 * Copyright Carnegie Mellon University, all rights reserved  *
 **********************************************************/
#include <math.h>

#define WGS84_a 6378137.0
#define WGS84_ee 0.00669437999013

 /* M_PI is already defined on the SGI, but not under Windows. */
```

```
#define M_PI        3.14159265358979323846

/* converting degrees to radians and vice versa */
#define DEG2RAD(DEG) ((double)(DEG)*M_PI/180.0)
#define RAD2DEG(RAD) ((double)(RAD)*180.0/M_PI)
```

Latitude and longitude are represented either as a single, floating point number representing degrees/radians, or as three separate numbers representing integer degrees, integer minutes, and floating point seconds. Latitude ranges from -90.0 to +90.0 degrees ($-\pi/2$ to $\pi/2$ radians); longitude from -180.0 to 180.0 ($-\pi$ to π radians), with negative numbers meaning "west" longitude, (west of the central meridian) and positive numbers meaning "east". Many instrument readouts give results in degrees, minutes, seconds notation. Here are some simple routines for converting between degrees/radians and degrees, minutes, seconds.

```
/* Convert degrees, minutes, seconds to floating point degrees */
double dms_to_deg(int d, int m, float s)
{
  double absd, deg;
  absd = fabs((double)d);
  deg = absd + (double)m / 60.0 + (double)s / 3600.0;
  if (d < 0) deg = - deg;
  return(deg);
}

/* Convert degrees, minutes, seconds to floating point radians */
double dms_to_rad(int d, int m, float s)
{
  return(DEG2RAD(dms_to_deg(d,m,s)));
}

/* Convert floating point degrees to degrees, minutes, seconds */
void deg_to_dms (double deg, int *d, int *m, float *s)
{
  double tmp, tmp2;
  int dd, mm;
  float ss;
  tmp = fabs(deg);
  dd = (int)tmp;
  tmp2 = (tmp - (double)dd) * 60.0;
  mm = (int)tmp2;
  tmp = (tmp2 - (double)mm) * 60.0;
  ss = (float)tmp;
  if (deg < 0) dd = (- dd);
  *d = dd; *m = mm; *s = ss;
  return;
}
```

```
/* Same as above, but output is a string suitable for printing */
void deg_to_dms_string (double deg, char *string)
{
  int d, m;
  float s;
  deg_to_dms(deg,&d,&m,&s);
  sprintf(string,"%d  %d  %.3f",d,m,s);
  return;
}
```

A local vertical coordinate system (LVCS) is a Cartesian coordinate system with its origin located somewhere at or near the site of interest. Point coordinates are represented as x,y,z distances, in meters, from the origin, where the coordinate system is oriented so that the positive X axis points east, positive Y points north, and positive Z points up (parallel to gravity). All that is needed to completely specify the LVCS is the 3D geodetic coordinate of the origin point. The following structure stores this point, along with several other values computed from it, such as rotation matrices, that are needed during the conversion to and from GPS and LVCS.

```
typedef struct {
   /* origin in lat, lon, (radians) and height (meters) wrt to WGS84   */
  double lat0rad, lon0rad, hmeters;
   /* origin in geocentric xyz.  Geocentric coords are defined wrt      */
   /* the center of the WGS84 ellipsoid.  X goes through the Greenwich */
   /* meridian, Z through the north pole, and Y is orthogonal to both. */
  double x0_ecef, y0_ecef, z0_ecef;
   /* rotation matrix to reorient local cartesian coord system so that */
   /* xy plane is tangent to WGS84 ellipsoid at origin lat0,lon0       */
  double rotmat[9];
  double invrotmat[9];
} LVCS;

/* Initialize the LVCS structure given a specific GPS origin point */
void init_lvcs(LVCS *lvcs, double lat0rad, double lon0rad, double hmeters)
{
  double x,y,z;
  lvcs->lat0rad = lat0rad;
  lvcs->lon0rad = lon0rad;
  lvcs->hmeters = hmeters;
  llh_to_ecef(lat0rad, lon0rad, hmeters, &x, &y, &z);
  lvcs->x0_ecef = x;
  lvcs->y0_ecef = y;
  lvcs->z0_ecef = z;
  comp_lvcs_rotmat(lvcs->rotmat, lat0rad, lon0rad);
 /* transpose matrix */
  lvcs->invrotmat[0] = lvcs->rotmat[0];
```

```
  lvcs->invrotmat[1] = lvcs->rotmat[3];
  lvcs->invrotmat[2] = lvcs->rotmat[6];
  lvcs->invrotmat[3] = lvcs->rotmat[1];
  lvcs->invrotmat[4] = lvcs->rotmat[4];
  lvcs->invrotmat[5] = lvcs->rotmat[7];
  lvcs->invrotmat[6] = lvcs->rotmat[2];
  lvcs->invrotmat[7] = lvcs->rotmat[5];
  lvcs->invrotmat[8] = lvcs->rotmat[8];
  return;
}
```

Two of the intermediate values needed during conversion, and that are stored above in the LVCS structure, are the coordinates of the origin in a "geocentric" coordinate system, and a rotation matrix that reorients geocentric coordinates into LVCS coordinates. Geocentric coordinates are defined with respect to a Cartesian coordinate system with its origin at the center of the WGS84 ellipsoid (i.e. the center of the Earth), oriented so that its X-axis goes through the Greenwich meridian, Z through the north pole, and Y is orthogonal to both. This is also called the Earth Centered Earth Fixed (ECEF) coordinate system. The following routines perform conversion of a geodetic point to an ECEF point, and compute the rotation matrix that reorients the ECEF coordinate axes to point along the LVCS axes.

```
/* Convert geodetic lat-lon-height to geocentric x-y-z */
/* note that lat and lon are specified in radians */
void llh_to_ecef(double latrad, double lonrad, double hmeters,
                 double *x, double *y, double *z)
{
   double tmp,N,h;
   h = hmeters;
   tmp = sin(latrad);
   N = WGS84_a / sqrt(1.0 - WGS84_ee*tmp*tmp);
   *x = (N + h)*cos(latrad)*cos(lonrad);
   *y = (N + h)*cos(latrad)*sin(lonrad);
   *z = (N*(1.0-WGS84_ee) + h)*sin(latrad);
   return;
}

/* Compute rotation matrix that reorients ECEF axes into LVCS axes */
void comp_lvcs_rotmat(double *m, double lat0rad, double lon0rad)
{
   m[0] = -sin(lon0rad);
   m[1] =  cos(lon0rad);
   m[2] = 0.0;
   m[3] = - sin(lat0rad) * cos(lon0rad);
   m[4] = - sin(lat0rad) * sin(lon0rad);
   m[5] = cos(lat0rad);
   m[6] = cos(lat0rad) * cos(lon0rad);
```

```
   m[7] = cos(lat0rad) * sin(lon0rad);
   m[8] = sin(lat0rad);
   return;
}
```

Here at last is the routine that converts from GPS coordinates to LVCS coordinates. The algorithm works by converting both the GPS point and the LVCS origin point into ECEF coordinates, subtracting them to find the vector between the LVCS origin and the GPS point, then rotating the vector so that its x,y,z coordinates are distances (in meters) from the LVCS origin to the GPS point, as measured along the LVCS X-Y-Z axes.

```
 /* convert between GPS coords and LVCS */
void gps_to_lvcs(LVCS *lvcs, double latrad, double lonrad, double hmeters,
                 double *xlvcs, double *ylvcs, double *zlvcs)
{
  double tmpxyz[3], lxyz[3];
  llh_to_ecef(latrad, lonrad, hmeters, &tmpxyz[0], &tmpxyz[1], &tmpxyz[2]);
  tmpxyz[0] -= lvcs->x0_ecef;
  tmpxyz[1] -= lvcs->y0_ecef;
  tmpxyz[2] -= lvcs->z0_ecef;
  mat_times_vec(lvcs->rotmat,tmpxyz,lxyz);
  *xlvcs = lxyz[0];
  *ylvcs = lxyz[1];
  *zlvcs = lxyz[2];
  return;
}
```

Conversion between LVCS and GPS coordinates is similarly performed through an intermediate ECEF conversion. Unfortunately, the algorithm for converting ECEF coordinates to geodetic lat-lon-height is not a simple closed form function, as it is in the other direction, but instead requires an iterative solution. The iteration algorithm is very well-behaved however, so should not be viewed as a drawback.

```
 /* 1.0e-7 gives better than .1 meter accuracy, at Pittsburgh's latitude */
#define DLATTHRESH 1.0e-7

 /* iterative solution to convert ECEF coords to geodetic coords */
void ecef_to_llh(double x, double y, double z,
 double *latrad, double *lonrad, double *hmeters)
{
   double lon, lat, d, newlat, dlat, N, tmp;
   lon = atan2(y,x);
   d = sqrt(x*x + y*y);
   lat = atan2(z,d);
   dlat = 1.0;
```

```
  while (dlat > DLATTHRESH) {
    tmp = sin(lat);
    N = WGS84_a / sqrt(1.0 - WGS84_ee*tmp*tmp);
    newlat = atan2((z + N * WGS84_ee*tmp),d);
    dlat = fabs(newlat - lat);
    lat = newlat;
  }
  *latrad = lat;
  *lonrad = lon;
  *hmeters = d / cos(lat) - N;
  return;
}

/* convert between LVCS and GPS coords */
void lvcs_to_gps(LVCS *lvcs, double xlvcs, double ylvcs, double zlvcs,
                 double *latrad, double *lonrad, double *hmeters)
{
  double tmpxyz[3], lxyz[3];
  lxyz[0] = xlvcs;
  lxyz[1] = ylvcs;
  lxyz[2] = zlvcs;
  mat_times_vec(lvcs->invrotmat,lxyz,tmpxyz);
  tmpxyz[0] += lvcs->x0_ecef;
  tmpxyz[1] += lvcs->y0_ecef;
  tmpxyz[2] += lvcs->z0_ecef;
  ecef_to_llh(tmpxyz[0],tmpxyz[1],tmpxyz[2],latrad,lonrad,hmeters);
  return;
}
```

Since LVCS is a locally defined system with respect to a particular origin point, it is possible that two people might have built models of the same site using different LVCS coordinate systems. That is, LVCS is not a globally agreed upon coordinate system definition, as the GPS coordinate system is. We present the following routine for converting points between two different LVCS coordinate systems. Conversion is performed using intermediate ECEF coordinates.

```
/* convert between two LVCS coordinate systems */
void lvcs1_to_lvcs2(LVCS *lvcs1, double x1, double y1, double z1,
    LVCS *lvcs2, double *x2, double *y2, double *z2)
{
  double tmpxyz[3], lxyz[3];
 /* convert from lvcs1 to ecef */
  lxyz[0] = x1;
  lxyz[1] = y1;
  lxyz[2] = z1;
  mat_times_vec(lvcs1->invrotmat,lxyz,tmpxyz);
  tmpxyz[0] += lvcs1->x0_ecef;
```

```
  tmpxyz[1] += lvcs1->y0_ecef;
  tmpxyz[2] += lvcs1->z0_ecef;
 /* convert from ecef to lvcs2 */
  tmpxyz[0] -= lvcs2->x0_ecef;
  tmpxyz[1] -= lvcs2->y0_ecef;
  tmpxyz[2] -= lvcs2->z0_ecef;
  mat_times_vec(lvcs2->rotmat,tmpxyz,lxyz);
  *x2 = lxyz[0];
  *y2 = lxyz[1];
  *z2 = lxyz[2];
  return;
}
```

Finally, for completeness' sake, we present the function `mat_times_vec` that has been used several times to multiply a $3x1$ vector by a $3x3$ matrix on the left.

```
void mat_times_vec(double *mat, double *vec, double *res)
{
   res[0] = mat[0] * vec[0] + mat[1] * vec[1] + mat[2] * vec[2];
   res[1] = mat[3] * vec[0] + mat[4] * vec[1] + mat[5] * vec[2];
   res[2] = mat[6] * vec[0] + mat[7] * vec[1] + mat[8] * vec[2];
   return;
}
```

E Ethernet communications in C++

This section contains some sample code for Ethernet communications. Newer versions of this code will be made available to FREs.

Instructions about using this code are as follows:

To use the "CDataLine" class, the user should declare a "CDataLine" object:

`private: CDataLine dl;`

Before actually using the Ethernet, the user should initialize it. The "CDataLine" class provides a member function to do this. The "Initialize" member function will return an error when the Ethernet can not be connected.

After correctly initializing the network, the user can send and receive packages using corresponding member functions of "CDataLine".

Currently, the available "CDataLine" member functions are:

`CDataLine();` Constructor function. Initialize internal parameters

`~CDataLine();` Destructor function. Close the network connection if the network is still open.

`bool Initialize( LPCTSTR lpszHostAddress, UINT nHostPort,int dWaitTime=0 );` Initialize the network.

`Bool Close();` Close the network

`bool ReceiveOCUData(OCU* p2);` Returns TRUE if correctly received a OCU package. Receiving is asynchronized.

`bool SendSPUData(SPU* p);` Returns TRUE if package is correctly sent. If the network is broken this member function will returns FALSE, so user should consider re-initialize the network.

`void PackSPUValues(SPU *p, char *packet);` Wrap up the SPU to network package

`void UnpackOCUValues(char *packet, OCU *p);` Unwrap the network package and write corresponding to the OCU object.

`bool IsDataAvailable();` Returns TRUE there's available data.

A piece of sample code using the dataline object looks like this:

```
OCU p2;
SPU p1;

if(!dl.Initialize(SERV_HOST_ADDR,SERV_TCP_PORT))
{
AfxMessageBox("Cannot open data line! Check it and play again", MB_OK |
MB_ICONEXCLAMATION);
}

else
{
//Set SPU Default Value, or to it by main loop
SetSPUDefault(&p1);

if(!dl.SendSPUData(&p1))
{
AfxMessageBox("Cannot send SPU data! Check it and play again",
MB_OK | MB_ICONEXCLAMATION);
}
DWORD starttime=GetTickCount();
while(GetTickCount()-starttime<1000);
if(!dl.ReceiveOCUData(&p2))
{
AfxMessageBox("Cannot receive OCU data! Check it and play again",
MB_OK | MB_ICONEXCLAMATION);
}

dl.Close();
```

```
}

In SPU the following structure is actually used:

/*While Initializing the system */
CDataLine* dl = new(CDataLine);
.
.
If(dl->Initialize(SERV_HOST_ADDR,SERV_TCP_PORT))
PROCESS THE ERROR;
.
.
SetTimer(NETWORK_TRANSFERING_TIMER_ID, TIMER_INTERVAL);

For example, in 1997 VSAM Demo the TIMER_INTERVAL was set to be 500ms;

/*While Timer time up*/
if(TimerID== NETWORK_TRANSFERING_TIMER_ID)
/* Since we may have multiple timers simultaneously */
{
if(THERE ARE SPU DATA TO BE SENT)
{
if(!dl.SendSPUData(&p1))
{
RECONNECT THE NETWORK OR REPORT ERROR;
}
}

RECEIVE OCU DATA;
}
```

E.1 dataline.h

```
/*********************************************************************
 * DATALINE.CPP *
 * Object for transfering data between SPU and OCU *
 * Authors: Yanghai Tsin and Raju Patil. *
 * Created Oct 27th, 1997 *
 * Copyright Carnegie Mellon University, all rights reserved *
 *********************************************************************/

/* This code is distributed to DARPA VSAM FRE contractors for use only
in DARPA VSAM research and applications. Violators will be prosecuted */

#ifndef CDATALINE_H
```

```
#define CDATALINE_H

#include "afxsock.h"
#include "math.h"

#define MAX_BUFFER_LENGTH 65536
#define SPU_PACK_INVALID 0xff


#define MAXLINE 512
#define PACKET_SIZE 500000
#define MAX_TARGETS 10
#define MAX_ANGLE   180.0 /* Max value of Pan Tilt FOVs etc */
#define MAX_ALT    9000.0 /* Mt Everest is 8848 mts :-)     */
#define MAX_INT16  (pow(2.0,15.0) - 1.0) /* Max value held in 16 bits */
#define MAX_INT32  (pow(2.0,31.0) - 1.0) /* Max value held in 32 bits */

typedef struct target_box {
    __int16 Top, Bottom, Left, Right;
} TARGET_BOX;

typedef struct spu_target {
    char ID; /* Target ID */
    TARGET_BOX Box; /* Target bounding box (Pixels) */
} SPU_TARGET;


typedef struct spu {

    char ID; /* SPU ID */
char    Status;
    __int32 Time; /* SPU packet timecode
     Time: mS */
    double Lat, Lon, Alt; /* Sensor GPS coordinates
     Lat: Degrees
     Lon: Degrees
     Alt: Metres */
    double Pitch, Yaw, Roll; /* Sensor attitude
     Pitch: Degrees
     Yaw: Degrees
     Roll: Degrees */
    double Pan, Tilt; /* Sensor PTU attitude
     Pan: Degrees
     Tilt: Degrees */
    double H_fov, V_fov; /* Sensor FOVs
     H_fov: Degrees
     V_fov: Degrees */
```

```
    __int16 X_size, Y_size; /* Image extent
     xsize: Pixels
     ysize: Pixels */
    char Number_of_targets; /* Number of targets currently
   tracked */
    SPU_TARGET *Target; /* Targets tracked */
} SPU;


typedef struct ocu {
     char        ID;                      /* SPU ID */

     char        Command;                 /* Command for SPU */

     char        Target_ID;                  /* Target ID if command
                                             refers to a target */
     double      Pan, Tilt;               /* Sensor PTU attitude
                                               Pan: Degrees
                                               Tilt: Degrees */
} OCU;


class CDataLine
{

public:
bool IsDataAvailable();
bool ReceiveOCUData(OCU* p2);
bool SendSPUData(SPU* p);
void PackSPUValues(SPU *p, char *packet);
void UnpackOCUValues(char *packet, OCU *p);
bool Close();
 ~CDataLine();
bool Initialize( LPCTSTR lpszHostAddress, UINT nHostPort,int dWaitTime=0 );
 CDataLine();
private:
SOCKET m_Sock;
char m_lpszSizeStr[10];
int SendBuffer(char* buffer, int dwSend);
int ReceiveToBuffer(char* buffer, int dwReceived);
char m_Buffer[MAX_BUFFER_LENGTH];
int m_nWaitTime;
bool m_bDatalineInitialized;
};
#endif
```

E.2 dataline.cpp

```
/*********************************************************************
 * DATALINE.CPP *
 * Object for transfering data between SPU and OCU *
 * Authors: Yanghai Tsin and Raju Patil. *
 * Created Oct 27th, 1997 *
 * Copyright Carnegie Mellon University, all rights reserved *
 *********************************************************************/

/* This code is distributed to DARPA VSAM FRE contractors for use only
in DARPA VSAM research and applications. Violators will be prosecuted */


#include "stdafx.h"
#include "dataline.h"

CDataLine::CDataLine()
{
m_bDatalineInitialized=FALSE;
m_nWaitTime=0;
}

bool CDataLine::Initialize( LPCTSTR lpszHostAddress, UINT nHostPort,  int dWaitTime)
{
m_nWaitTime=dWaitTime;


if(m_bDatalineInitialized)
return TRUE;

   WSADATA WSAData;
   LPWSADATA lpWSAData;
   struct sockaddr_in sai;

   lpWSAData = &WSAData;

   /*build network address */
    sai.sin_family = AF_INET;
    sai.sin_addr.s_addr = inet_addr(lpszHostAddress);
    sai.sin_port = htons(nHostPort);

    WSAStartup(0x0101,lpWSAData);



   /*create the socket and bind the address */
```

```
    if ((m_Sock = socket( AF_INET, SOCK_STREAM, 0)) == -1)
       return FALSE;
    if (connect(m_Sock,(struct sockaddr *)&sai, sizeof(sai)) != 0)
       return FALSE;

//set socket to nonblocking mode
u_long ul=1;
ioctlsocket(m_Sock,FIONBIO, &ul);

m_bDatalineInitialized=TRUE;
return TRUE;

}

CDataLine::~CDataLine()
{
if(m_bDatalineInitialized)
Close();
}


bool CDataLine::Close()
{
closesocket(m_Sock);
m_bDatalineInitialized=FALSE;
return TRUE;
}

int CDataLine::ReceiveToBuffer(char * ptr, int nbytes)
{
int nleft, nread;

if(!m_bDatalineInitialized)
return FALSE;

   nleft = nbytes;

   while(nleft > 0)
   {
      nread = recv(m_Sock,ptr,nleft,0);

      if(nread < 0)
        return (nread);           /* Error, return < 0 */
      else if(nread == 0)
        break;                    /* EOF               */
```

```
      nleft -= nread;
      ptr += nread;
   }
   return(nbytes - nleft);        /* Return >= 0       */
}

int CDataLine::SendBuffer(char * ptr, int nbytes)
{
   int nleft,nwritten;
   nleft = nbytes;

   if(!m_bDatalineInitialized)
return FALSE;

   while(nleft > 0)
   {
      nwritten = send(m_Sock,ptr,nleft,MSG_DONTROUTE);

      if (nwritten <= 0)
         return(nwritten);   /* Error */

      nleft -= nwritten;
      ptr += nwritten;
   }
   return(nbytes - nleft);
}

void CDataLine::PackSPUValues(SPU *p, char *packet)
{

 int i,j,index;
 __int32 int32array[12];

 int32array[1] = p->Time;    /*Nothing stored in int32array[0]*/
 /*pack double values into ints */
 int32array[2] = (__int32)( (p->Lat/MAX_ANGLE)*MAX_INT32);
 int32array[3] = (__int32)( (p->Lon/MAX_ANGLE)*MAX_INT32);
 int32array[4] = (__int32)( (p->Alt/MAX_ALT)*MAX_INT32);

 int32array[5] = (__int32)( (p->Pitch/MAX_ANGLE)*MAX_INT32);
 int32array[6] = (__int32)( (p->Yaw/MAX_ANGLE)*MAX_INT32);
 int32array[7] = (__int32)( (p->Roll/MAX_ANGLE)*MAX_INT32);

 int32array[8] = (__int32)( (p->Pan/MAX_ANGLE)*MAX_INT32);
 int32array[9] = (__int32)( (p->Tilt/MAX_ANGLE)*MAX_INT32);
 int32array[10]= (__int32)( (p->H_fov/MAX_ANGLE)*MAX_INT32);
 int32array[11]= (__int32)( (p->V_fov/MAX_ANGLE)*MAX_INT32);
```

```
/*  The other values (ID, Time, X_size, Y_size and Number_of_targets  */
/*  are already int values (either __int32 or __int16)                */

/* Put ID in packet[0]      */
packet[0]= p->ID;
packet[1]= p->Status;

/* Put all int32array elements into packet                    */
/* the LSByte of int32array[0] should occupy packet[1]        */
/* the next LSByte of int32array[0] should occupy packet[2]  */
/* the next LSByte of int32array[0] should occupy packet[3]  */
/* the next LSByte of int32array[0] should occupy packet[4]  */
/* Then: the LSByte of int32array[1] should occupy packet[5] */
/* and so on ......                                           */
for(i=1;i<=11; i++)
   {
   for(j=0;j<4;j++)
      {
      index = j+(i-1)*4+2;
      packet[index]= (int32array[i] & (0xFF << (8*j))) >> (8*j);
      }
   }

/* Put remaining items of spu data into packet              */

 packet[++index]= (p->X_size) & (0xFF);
 packet[++index]= ((p->X_size) & (0xFF << 8)) >> 8;
 packet[++index]= (p->Y_size) & (0xFF);
 packet[++index]= ((p->Y_size) & (0xFF << 8)) >> 8;
 packet[++index]= p->Number_of_targets;

/* Put all values corresp to each target into packet */
/* Note: there are 9 bytes for each target           */

 for(i=0;i<(p->Number_of_targets);i++)
    {
    packet[++index] = p->Target[i].ID;
    packet[++index] = (p->Target[i].Box.Top) & (0xFF);
    packet[++index] = ((p->Target[i].Box.Top) & (0xFF << 8)) >> 8;
    packet[++index] = (p->Target[i].Box.Bottom) & (0xFF);
    packet[++index] = ((p->Target[i].Box.Bottom) & (0xFF << 8)) >> 8;
    packet[++index] = (p->Target[i].Box.Left) & (0xFF);
    packet[++index] = ((p->Target[i].Box.Left) & (0xFF << 8)) >> 8;
    packet[++index] = (p->Target[i].Box.Right) & (0xFF);
```

```
     packet[++index] = ((p->Target[i].Box.Right) & (0xFF << 8)) >> 8;
     }

}

void CDataLine::UnpackOCUValues( char *packet, OCU *p)
{
 int i,j,index;
 __int32 int32array[12],temp;
 index = 0;

 p->ID        = packet[0];
 p->Command   = packet[1];
 p->Target_ID = packet[2];

 /* The next 2 values (4bytes each) are extracted from packet  */
 for(i=1;i<=2;i++)
    {
    int32array[i] = 0;
    for(j=0;j<4;j++)
       {
       index = j+(i-1)*4+3;
   temp= (BYTE)packet[index];
       int32array[i] += temp << (8*j);

       }
    }

 /* The int value of the Pan and Tilt values are converted to their   */
 /* corresp float values and stored in the structure                  */
 /*pack double values into ints */

 p->Pan   = (((double)int32array[1])/MAX_INT32)*MAX_ANGLE;
 p->Tilt  = (((double)int32array[2])/MAX_INT32)*MAX_ANGLE;

}

bool CDataLine::SendSPUData(SPU * p1)
{
long spu_nbytes;
int nbytes;

if(!m_bDatalineInitialized)
return FALSE;

spu_nbytes = 51 + (  (p1->Number_of_targets) * 9);
```

```
PackSPUValues(p1,m_Buffer);


    sprintf(m_lpszSizeStr,"%d",spu_nbytes);
    nbytes = sizeof(m_lpszSizeStr);
    if(SendBuffer(m_lpszSizeStr,nbytes)!=nbytes)
return FALSE;


    if(SendBuffer(m_Buffer,spu_nbytes)!=spu_nbytes)
       return FALSE;

return TRUE;
}

bool CDataLine::ReceiveOCUData(OCU * p2)
{
long ocu_nbytes;
int nbytes;

if(!m_bDatalineInitialized)
return FALSE;

if(!IsDataAvailable())
return FALSE;
    nbytes = sizeof(m_lpszSizeStr);

if(ReceiveToBuffer(m_lpszSizeStr,nbytes)!=nbytes)
return FALSE;
    ocu_nbytes = atol(m_lpszSizeStr);

    if(ReceiveToBuffer(m_Buffer,ocu_nbytes)!=ocu_nbytes)
return FALSE;
    UnpackOCUValues(m_Buffer,p2);

return TRUE;

}



bool CDataLine::IsDataAvailable()
{
if(!m_bDatalineInitialized)
return FALSE;

if(recv(m_Sock,m_lpszSizeStr,sizeof(m_lpszSizeStr),MSG_PEEK)==SOCKET_ERROR)
```

```
return FALSE;

return TRUE;

}
```

- END OF APPENDICES -