FFI RAPPORT

MODULAR DISTRIBUTED SIGNAL PROCESSING NETWORK FOR CHESS

SAGVOLDEN Geir

FFI/RAPPORT-2000/01294

FFIE/711/116

Approved
Kjeller 12 july 2000

Stian Løvold
Director of Research

MODULAR DISTRIBUTED SIGNAL PROCESSING NETWORK FOR CHESS

SAGVOLDEN Geir

FFI/RAPPORT-2000/01294

FORSVARETS FORSKNINGSINSTITUTT
Norwegian Defence Research Establishment
P O Box 25, NO-2027 Kjeller, Norway

FORSVARETS FORSKNINGSINSTITUTT (FFI)
Norwegian Defence Research Establishment

UNCLASSIFIED

P O BOX 25
N0-2027 KJELLER, NORWAY

SECURITY CLASSIFICATION OF THIS PAGE
(when data entered)

REPORT DOCUMENTATION PAGE

1) PUBL/REPORT NUMBER FFI/RAPPORT-2000/01294 1a) PROJECT REFERENCE FFIE/711/116	2) SECURITY CLASSIFICATION UNCLASSIFIED 2a) DECLASSIFICATION/DOWNGRADING SCHEDULE -	3) NUMBER OF PAGES 37

4) **TITLE**
MODULAR DISTRIBUTED SIGNAL PROCESSING NETWORK FOR CHESS

5) **NAMES OF AUTHOR(S) IN FULL (surname first)**
SAGVOLDEN Geir

6) **DISTRIBUTION STATEMENT**
Approved for public release. Distribution unlimited. (Offentlig tilgjengelig)

7) **INDEXING TERMS**

	IN ENGLISH:		IN NORWEGIAN:
a)	Signal processing	a)	Signalbehandling
b)	Computer software	b)	Programvare
c)	Object oriented programming	c)	Objektorientert programmering
d)		d)	
e)		e)	

THESAURUS REFERENCE:

8) **ABSTRACT**

This report documents the modular distributed signal processing system developed for the CHESS project. The system is made of small modules, each performing a signal processing, data visualisation or file access task. System functionality is defined by piping the data stream between modules, making a data flow tree. The common library defining the communication interface and the data structure for all modules, libCHESS, is described in detail. The modules are described briefly, as well as modules planned for implementation in the future

9) DATE	AUTHORIZED BY This page only	POSITION
12 july 2000	Stian Løvold	Director of Research

ISBN-82-464-0456-3

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE
(when data entered)

CONTENTS

MODULAR DISTRIBUTED SIGNAL PROCESSING NETWORK FOR CHESS

1 INTRODUCTION

This document describes the data processing network developed as a part of the *Composite Hull Embedded Sensor System* (CHESS) project. CHESS is a joint project of the Norwegian Defense Research Establishment (FFI), Kjeller, Norway and the Naval Research Labs (NRL), Washington D.C., USA. The project goal is to establish techniques for continuous surveillance of composite hull structures using fiber optic Bragg gratings as primary sensors(1, 2).

A part of this work has been to develop computer programs for the continuous online processing and analysis of sensor data. An important aspect of a surveillance system is the ability to give instantaneous feedback of the current load to the ship’s operator, allowing him to maximize performance while reducing the risk of structural damage. In addition, data may be logged for subsequent analysis.

Data are passed from the data source to the data sink as a continuous stream in most surveillance systems. In CHESS, the primary data sources are strain data measured at several locations on the ship’s hull. Although such local measurements may give an indication of the total load sustained by the hull, transformations involving data from several sensors are necessary for a proper estimate. Thus, several operations, such as filtering, downsampling and transformations, are performed on the data before they are presented or saved.

These operations are implemented as separate programs accepting a stream of data packets from a server program, while supplying processed data to one or more client programs. These *signal processing modules* all employ a common data interface and –structure. Most accept several data formats, and may be connected in any order. Thus, a signal processing task is approached by deciding which operations are to be performed on the data, and then connecting signal processing modules in the corresponding sequence.

Data communication between modules are based on TCP/IP sockets, allowing data to be passed between modules running on different computers in a network. This allows us to distribute the computation load on several computers to perform heavy signal processing tasks, but also to present data on computers located elsewhere, for instance in the machine control room or on the bridge. The protocol supports connection and disconnection of modules during operation, thus allowing the processing network to change in response to demand.

The current software is implemented on PC platforms running the Linux operating system. This platform was chosen due to the availability of a real-time operating system, RT-Linux, necessary for the data collection program (3).

This report is divided in four major parts. First, an overview of existing and proposed signal processing networks is given to provide a motivation for the technical discussions to follow. Then, the architecture of a generic signal processing module is outlined, followed by a detailed description of *libCHESS*, a software library of routines common to the signal processing modules. The aim is to provide a sufficient background for programmers intending to develop modules for the system. Finally, the modules developed for the CHESS project are documented.

2 SIGNAL PROCESSING NETWORKS

Elements in a signal processing network may be roughly classified as data sources, data operators and data sinks. A data source is the start of a data stream and may be a program distributing data from one or more sensors, or playing back recorded data for subsequent analysis or test. A data operator performs an operation on the data, such as transformations or filtering, before passing the data on. A data sink is the ending point of a data stream. Typical data sink operations may be to log data to disk or display them on a screen.

Several processing networks have been designed during the CHESS project, and a few of them is presented in this report for illustration purposes.

2.1 Bending moment calculation and display

In Figure 2.1, a network is shown that displays the bending moment while logging raw data to disk. The network was used for monitoring bending moments during cannon firing tests on board KNM Skjold(4).

Following data collection, a 5 mHz high-pass elliptic filter removes slowly varying components from the measured local strains. These signals may be due to unwanted temperature-induced drift, or re-distribution of the hull load due to fuel consumption or changes in cushion pressure, but cannot be separated in the current sensor implementation.

After high-pass filtering, the data are downsampled and low-pass filtered to reduce computational load. The 13Hz lowpass filter is included to filter out contributions from local vibration modes in the panels the sensors are affixed to. Finally, information from several sensors is combined to estimate the hull load which is then displayed on the screen.

2.2 Continuous surveillance – Spring 2000

A test system for continuous load monitoring was installed onboard KNM Skjold in January 2000. This system is intended to serve as a “flight-recorder” system operated by the KNM Skjold crew. The system contains three logging applications; one continuous logger of raw data for backup purposes and post-processing, one application logging the extremal

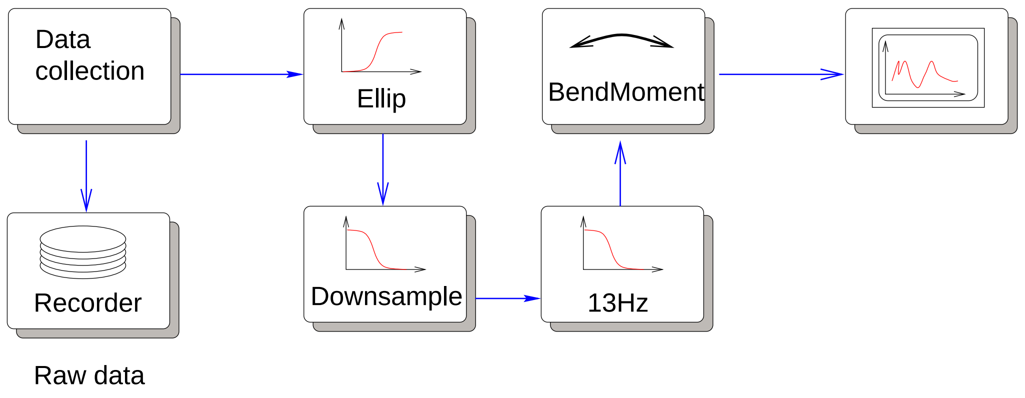


Figure 2.1 *A simple signal processing network for calculating bending moments and displaying them on the screen. In addition, data are logged for post-processing*

points of the calculated ship load curve, and one application logging timestamps of events of high local loads to facilitate post-processing. In addition, graphics showing the instantaneous load and a 30 minute load history is presented to the operator.

To facilitate operation, the data stream to the signal processing network in Figure 2.2 may be switched on and off using the *DataSwitch* application. Drift is removed from the data using a 5 mHz elliptic filter. The top network arm logs the times when the local strain exceeds a pre-defined level, currently 700 $\mu\epsilon$. The middle arm downsamples the calculated bending moments twice before the data are passed through a 1.75 Hz lowpass filter. This frequency is below the lowest ship eigenfrequency, thus the output from the filter bank is the bending moments directly induced by the waves.

Extremal points in the wave-induced bending moments are found from changes in the sign of the derivative, and logged to disk. The estimated bending moments are also plotted to screen to give the ship's operator instantaneous information on the current sea loads.

The lower arm of the signal processing network saves raw data, downsampled by a factor 4, to disk.

2.3 Future signal processing networks

A wide variety of signal processing applications may be assembled using the building blocks already developed for CHESS, and new modules will be developed in accordance with the needs of the project. However, some modules are already planned and described in chapter 6.

These modules focus on providing additional data input to the network and implementing advanced signal processing applications and modeling to further optimize the feedback and

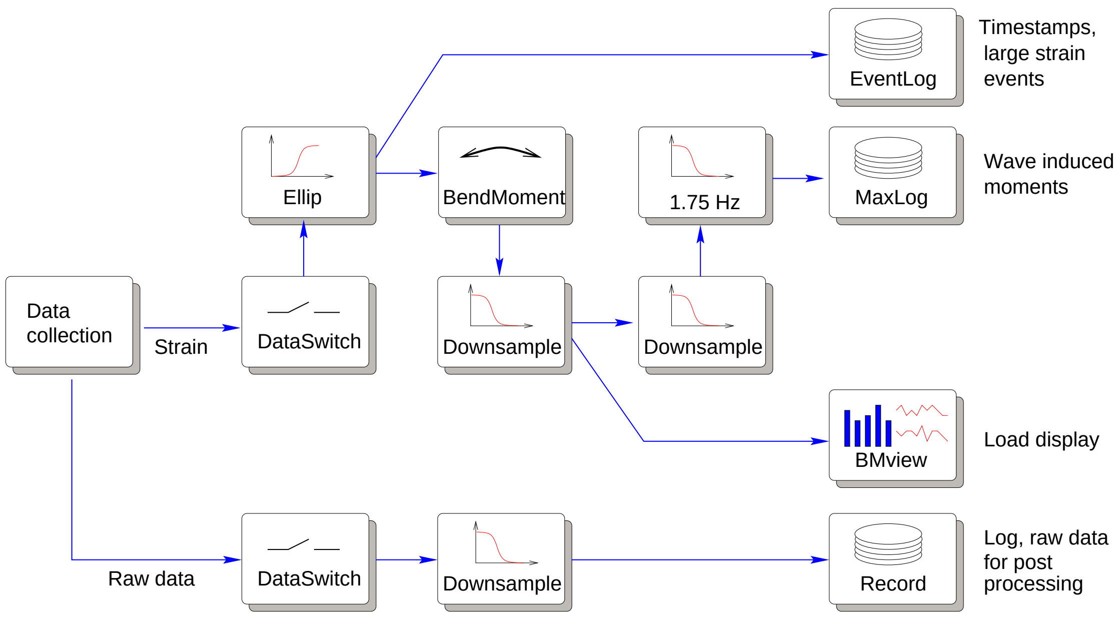


Figure 2.2 *The signal processing network used for the permanent data logging application during Spring 2000.*

recommendations output from the surveillance system to the operator of the ship.

On the data acquisition side, a module accepting data from instruments communicating using the *NMEA 183* serial interface protocol is a high priority. This protocol is implemented in several marine instruments, such as GPS receivers.

Also, the ability to receive data from instruments giving voltage level output, such as the amplified signal from accelerometers and conventional strain gauges, is important. A generic driver for A/D cards is therefore planned.

The possibility of exchanging data with the ship systems will also be explored. This would enable the data processing filters to process information on the current operation mode, which, in the long run, may result in better characterization of the ship's performance under various sea conditions and operation modes, leading to better feedback to the operator.

Including several sources in the signal processing system may lead to heterogeneous data matrices and sampling rates. An application for merging data from various sources should therefore be implemented.

On the signal processing side, several of the algorithms used in the post processing of data may be implemented as transformations on the data. These include calculation of local strain levels and directions at sensor rosettes and comparison of the local strain levels to the material strength using the Tsai Wu Last Ply Failure criterion. Wavelet and short time Fourier transform modules should also be developed for both filtering and visualization purposes.

The amount of data presented to the ship operator, and presentation form, is an important consideration in visualization modules. The amount of information available is far greater than what is necessary for proper operation of the ship. Clearly, the ideal form of presentation would be messages or graphics drawing attention to sub-optimal operation settings and operation modes that could pose a danger to the integrity of the hull.

3 THE SIGNAL PROCESSING MODULE

This chapter describes the design of a generic signal processing module. Emphasis is put on the common features of all modules, and the use of a common library of functions. This library, libCHESS, is described in detail in the next chapter.

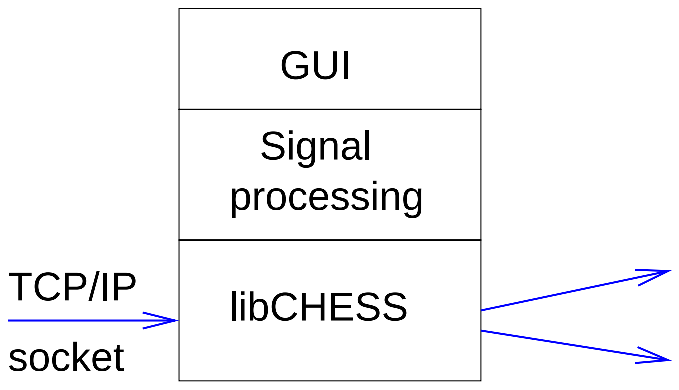


Figure 3.1 Layers in the generic signal processing module

The signal processing module may be described as having three layers. The bottom layer contains methods handling the data structure and the data stream. This layer includes functions for receiving data from another module, accessing the data structure, and transmitting processed data to subscribing modules. The middle layer consists of the functions performing operations on the data, while the top layer consists of the graphical interface presented to the user.

All programs developed for the CHESS signal processing system are based on the Qt toolkit from Troll Tech A/S (5). This is a toolkit for making event driven programs. An event driven program is a program where execution is controlled by a table connecting a number of defined events, such as the user clicking a specific button, with program parts that should be executed following such an event. The toolkit also contains a number of standard graphical elements, such as buttons, sliders etc, and all graphics presented to the user is based on this toolkit.

However, in the context of signal processing modules, the most important event is receiving data. The Qt toolkit will listen for new data from the data source, and run the processing routine once data are received.

This programming style is lean, as a module only consumes processing power when it receives data.

3.1 The data stream

The CHESS data structure consists of a data buffer and identifiers defining its contents. The data have several properties that are assumed to be constant. These are related to the number of sensors in the data stream, the type of data (i e integer, float etc), the names of the sensors etc. When a connection between CHESS modules is established, this information defining the data stream is transmitted from the data source to the client. The client should then inspect the information and decide whether it can handle the data offered.

When data are received,they are available in a data input buffer. Using the information on the data format, the module processes the data and copies the results to an output buffer. The results are then transmitted to the module's clients.

3.2 Socket communication

CHESS modules communicate using TCP/IP sockets. A socket is a protocol for transferring data between two computers on the Internet, and is widely used in mail servers, Internet browsers, remote login programs and for file transfers. Thus, socket communication is a widely accepted standard which is available for most operating systems.

A socket connection is uniquely defined by the name of the computer and the port number. A module is thus started with arguments defining the hostname and port number of the data source and the port number its clients may connect to.

A module does not accept connections on the send port until a connection is established with the data source. When the data source disconnects, all connections on the send port are terminated.

A module will attempt to connect to a data source at a fixed interval[1]. Also, the module will repeatedly try to allocate the send port if it was initially unavailable. Thus, a module may be started before the intended source module is running, and may reconnect if the source module is restarted. Also, clients may connect and disconnect at any time while the send port is available.

3.3 Inheritance

The main purpose of the signal processing module is to perform mathematical operations on data in a buffer. Thus, the way data are moved into the input buffer, and transmitted from

[1]The retry period is currently 1s

the output buffer, is not a concern for the processing algorithm. The data buffers are therefore built as outlined in the class inheritance tree shown in Figure 3.2.

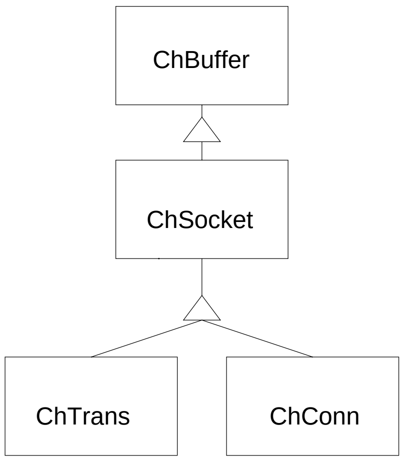


Figure 3.2 Classes derived from the basic ChBuffer class

Here, the class ChBuffer contains the data buffer and all information and methods necessary to access it. ChSocket adds functionality for socket communication, while ChTrans and ChConn add methods for repeated connection attempts and handling of connection and disconnection events for a transmitting and receiving connection, respectively.

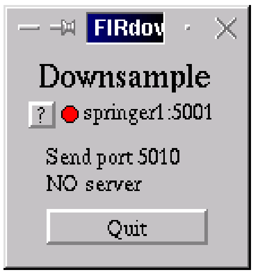
Thus, a module would construct ChTrans and ChConn –type variables for their transmitting and receiving buffers, but access these buffers with pointers cast to the ChBuffer class. This allows several modules to be merged into a larger module at a later stage, bypassing socket communication, leading to higher data throughput in an optimized application.

3.4 The user interface

The user interface of the typical module only contains status displays and a “Quit” button for closing down the module. As shown in Figure 3.3, the first display contains information on the link to the data server. The button marked “?” brings up information on the data, while the red circle becomes green when the connection to the server is made.

Below, the send port number and the status of the transmitting server is displayed. The message “NO server” indicates that the send port is not active, which is the case when no data is received. When the send port is active, the second line in this display will indicate the number of clients connected.

These message displays are implemented in the common library as the ChConnStatus and ChSendStatus classes.



Downsample
springer1:5001
Send port 5010
NO server

Figure 3.3 The user interface for a typical module

3.5 Setting up the network

Signal processing modules are started with command line options detailing which port on what computer it receives data from, the send port for the processed data and other module-dependent options.

This interface was chosen instead of a graphical one to allow the user to make command scripts starting a large number of modules by entering a single command. The signal processing network is therefore usually set up as commands in a UNIX shell script, detailing all connections and input files.

The user may, of course, also start individual modules connecting to the network anytime.

4 LIBCHESS IMPLEMENTATION

This chapter documents the implementation of the main components of libCHESS, and aims at providing the background information enabling CHESS programmers to maintain and extend the library and use it when creating new CHESS signal processing modules.

However, a detailed description of every aspect of the library functions will not be given. For this the programmer is referred to the source code and comments therein.

4.1 A brief introduction to the Qt library

All modules use the Qt library by Troll Tech(5) for most graphical elements and event driven program flow control. A CHESS programmer should consult the documentation provided with this package for details, however a short introduction to the Qt signal/slot mechanism is provided below for the reader’s convenience.

Consider a program presenting buttons “A” and “B” to the user. The user may, using the mouse, press either button, or no button at all. The program has to wait for the mouse to be clicked, check if the mouse cursor was inside the button, and perform the task selected.

```
#!/bin/tcsh

# Set path defaults

setenv BINDIR "/usr/local/CHESS/bin"
setenv DATADIR "/ffi/bonde1/proj2/SeaLog"
setenv INPUTDIR "$DATADIR/inputs"
setenv DHOST "bonde1"

kstart "$BINDIR/DataSwitch -f $INPUTDIR/SeaLog.SWF" -desktop 1
kstart "$BINDIR/Ellip -h $DHOST -p 5003 -f $INPUTDIR/5mHz_1ell.FLT -s 5011" -desktop 2
kstart "$BINDIR/LaserAlarm -d 2 -h $DHOST -p 5002 -f $DATADIR/logs/SeaLog_Alarms.txt" -desktop 1

# Raw Data branch

kstart "$BINDIR/Downsample -h $DHOST -p 5012 -f $INPUTDIR/lowpass1.FIR -s 5013" -desktop 2
kstart "$BINDIR/AutoRecord -h $DHOST -p 5013 -f $DATADIR/raw/b1raw" -desktop 1

# BendMoment branch and filterbank

kstart "$BINDIR/BendMoment -h $DHOST -p 5011 -f $INPUTDIR/bendmoment_ACBD.MTX -s 5020" -desktop 2

kstart "$BINDIR/Downsample -h $DHOST -p 5020 -f $INPUTDIR/lowpass1.FIR -s 5021" -desktop 2
kstart "$BINDIR/Downsample -h $DHOST -p 5021 -f $INPUTDIR/lowpass1.FIR -s 5022" -desktop 2
kstart "$BINDIR/Downsample -h $DHOST -p 5022 -f $INPUTDIR/low175_bank.FIR -s 5023" -desktop 2

kstart "$BINDIR/MaxLog -h $DHOST -p 5023 -f $DATADIR/maxlog/b1mBM" -desktop 1

# Event logging

kstart "$BINDIR/EventLog -h $DHOST -p 5011 -f $DATADIR/logs/SeaLog_Events.asc -n 1000" -desktop 1

# Viewers

kstart "$BINDIR/BMview -h $DHOST -p 5021 -c $INPUTDIR/BMview.conf" -desktop 1
kstart "$BINDIR/MaxMin -h $DHOST -p 5021" -desktop 1
```

Table 3.1 The script file staring the network shown in Figure 2.2, used during the long–term logging test spring 2000. kstart is a utility to control the location of the module's graphical user interface window

A simplistic (and extremely inefficient) program would use an endless loop to check whether the mouse was pressed or not. This loop may consume all system resources, slowing down the execution of other processes.

The Qt library hides these implementation details from the programmer. Here, the mouse click is considered an event, and the library maintains a table of functions to be run following each event.

Qt is an object oriented program in C++ and contains a lot of standard objects, such as graphical buttons. The simple program described above would then be made using two button objects. When a button object notices it has been clicked, it emits a *signal*. Qt checks whether this signal has been connected to a function to be run, called a *slot*. Program flow control in Qt is therefore set up by associating signals and slots using the connect statement.

All classes derived from the QWidget baseclass may emit signals and provide slots. The user is, therefore, not limited to the signals defined by the Qt library, but may extend the signal set to fit his/her needs.

4.1.1 Signal/slot mechanism in CHESS modules

CHESS modules use the Qt signal/slot mechanism extensively for program control. The most important is the *NewData()* signal defined in the ChBuffer class. This signal is emitted when new data are received into the buffer. The CHESS module programmer connects a slot function to this signal, which is then run by Qt every time new data are received. The slot function performs the actual operations on the data.

4.2 CHESS data structure and communication tools

4.2.1 ChBuffer

ChBuffer is a fixed size first-in-first-out (FIFO) buffer holding CHESS data. The class also contains methods necessary for adding, accessing and removing data, as well as methods to access identifiers defining the data stream.

A CHESS data stream is organized as a sequence of data records. Each data record usually contains a timestamp and a number of data cells containing one or more data entries each. Usually, only one data entry, e.g. one strain value, is entered in each data cell. [1]

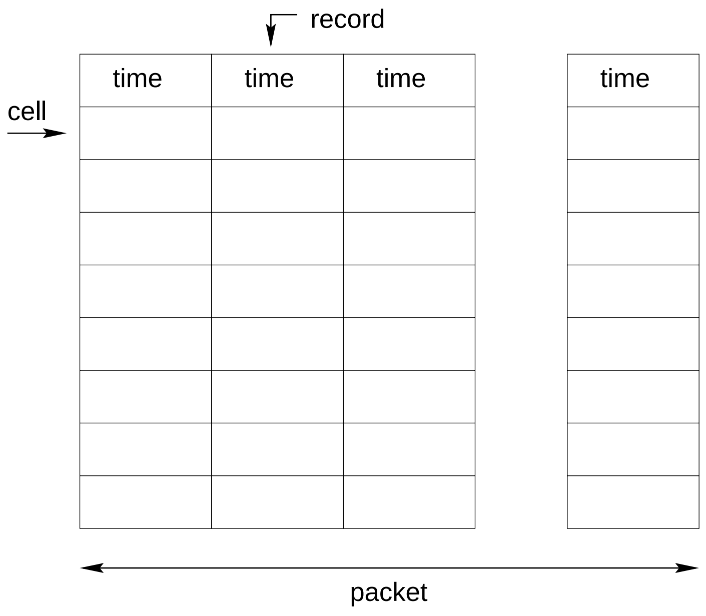


Figure 4.1 The structure of the CHESS data stream

At construction the ChBuffer data is undefined. The BufInit method takes parameters that define the organization of the data in the buffer, such as the data type and number of cells

[1]The possibility to allow several data entries in a cell was included to maintain a consistent data structure for transformations adding an extra dimension to the data, for instance short time Fourier transforms.

and buffer length (see tables 4.1 and 4.2). The buffer length may be changed using the BufResize method.

The ChBuffer may be accessed after initialization to set name labels on each cell (SetCellName), and to add blocks additional information (AddAIBlock).

Name	type	comment
SourceID	int	source identification
FlTime	bool	flag if data contains timestamp
DataID	int	data type identifier
id	int	internal identification number to identify the buffer, sent in NewData(int) signals
CellSize	int	number of data entries in each cell
NoCells	int	number of data cells for each record
PacketLength	int	number of records transmitted per packet (ChSocket)
CellName	char[M*N]	strings describing the data in each cell
AINum	int	number of additional info blocks
AISize	int[N]	size of each add. info block
AIBlock	char[..]	add. info block contents

Table 4.1 CHESS data header specifying the data stream

ID	type
1	char
2	unsigned short
3	short
4	unsigned int
5	int
6	unsigned long
7	long
8	float
9	double

Table 4.2 CHESS data identifiers

The CHESS data structure is, as any multi-dimensional data structure, really a one-dimensional array of bytes where the specifications of the data structure determines how the individual data records are accessed (Figure 4.2). To make data access as fast as possible, CHESS modules are implemented by directly accessing the data buffer. This is done by obtaining a pointer to the header of the data structure using the BufStart() function, and using information on the data organization from the data header to access the data. Thus, a typical data processing function would have the following structure

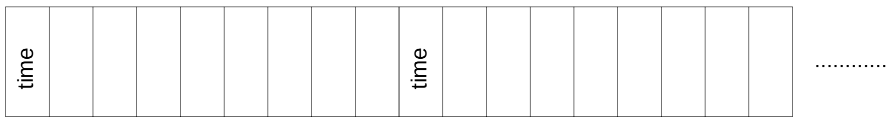


Figure 4.2 The organization of the data buffer

```
calculate the number of data variables to process

obtain pointer to data head, initialize variables

<loop1 : for all records>
  <loop2: for all cells in record>

    make calculation
    push results to output buffer
    point to next data variable

  <end loop2>
<end loop1>

call input buffer’s BufBlockRepos(..)
call output buffer’s EmitNewData()
```

The BufBlockRepos() function flushes the processed data from the input buffer, the argument is the number of blocks to flush, while EmitNewData() emits the output buffers NewData() signal, upon which Qt launches processes connected to this signal.

The buffer organization allows the processing algorithm direct random access to the data buffer, and reduces the number of data copy operations to a minimum.

4.2.2 ChSocket – Socket communication

ChSocket adds TCP/IP socket communication functionality to the ChBuffer class. A ChSocket may either be a receiving socket or a transmitting socket. The mode is determined by the overloaded constructor.

Transmitting mode

A transmitting ChSocket is constructed with full details of the desired data structure. The constructor initializes its inherited ChBuffer, which will hold the data to be sent. The constructor is also supplied with the desired length of the packets the data stream is broken into.

After constructing the transmitting ChSocket, setting sensor names etc., Listen() is called to wait for connection attempts. Several clients may connect to a single ChSocket, each receiving the same data stream.

Data may be pushed to the buffer either by accessing the buffer directly and calling EmitNewData() when data are ready to be transmitted, or by calling the Write(char *message,int sendsize) function. The latter will transmit the data when the desired data packet size has been reached. The transmitted data packet is subsequently removed from the buffer.

Receiving mode

A receiving ChSocket will receive the details of the data structure upon successful connection to a transmitting ChSocket, the server. Thus, the constructor is supplied with bufmult, a data buffer multiplier, setting the “safety margin” to bufmult times the packet size. A buffer twice the packet size is normally recommended to avoid buffer overruns, and is the default. However, some signal processing tasks, such as FIR filters, need to keep a certain backlog of data and thus use a larger buffer.

Signals and slots

ChSocket has two signals. ClientConn(int) is emitted from a transmitting ChSocket whenever a client connects, the argument is the client number. Disconnectsignal() is emitted from a receiving ChSocket when the server disconnects.

At construction, the NewData signal of a transmitting socket is connected to the Send() slot. Thus, whenever the ChBuffer’s NewData signal is emitted, the ChSocket will transmit the data in the buffer to its clients.

4.2.3 ChTrans and ChConn

The implementation details of the ChSocket class is hidden from the programmer using the ChTrans for constructing a transmitting socket, and ChConn for constructing a receiving socket. These sockets add timers for repeated connection attempts to the ChSocket class, allowing the socket to connect at a later time if the initial connection attempt was rejected.

For the transmitting socket, the primary cause of rejection is failure to bind the requested socket. This is usually due to an operating system latency in freeing a previously bound socket, for instance when a module using a socket is stopped and immediately restarted. Experience with Linux sockets indicate that it may take several seconds before the socket is free. Another source of failure is, of course, cases when two modules attempt to use the same transmitting socket.

The receiving socket will fail connection attempts when the server socket is not available, e g when the server module is not started or has not successfully bound the socket.

Signals and slots

ChConn has two signals. NewData(int) is emitted when new data are received, the argument is the optional id number assigned at construction. This is useful for identifying the source of a NewData signal in a module receiving data on multiple sockets.

The ConnChange(bool) signal is emitted when the connection status changes, that is when a successful connection to the server is made, or the connection is lost. The argument is “true” when a connection is made.

ChTrans adds no signals or slots to the ChBuffer and ChSocket base classes.

4.2.4 Performance details

ChSocket utilizes the operating system functions made for transmitting data across the network, or locally. Thus, the data must be piped through the operating system, which involves several copying operations.

A test of transfer speed and reliability was carried out using a transmitting module which sent data packets at maximum speed. All packets had a unique identification number. The receiving module checked this number and the packet size, while calculating the sustained transfer rate.

The tests show that a ChSocket may transfer data reliably at a rate of 50 MB/s locally, and at almost 12 MB/s across a 100 Mbit/s network, see Figure 4.3. The peak rate for local transfers occurs at packet sizes somewhat below 64k, while for network transfers the rate is constant for packet sizes above 2k.

The local transfer rate is probably limited by the efficiency of the local memory manager, while the network transfer rate is close to the network bandwidth at 100 Mbit/s = 12.5 MB/s.

The receiving module also monitored the size of the packets received. For local transfers, packets less than about 50k were received as a single packet, while larger packets were broken up. This agrees with the maximum local transfer rate observed at this packet size. The packets were divided into packets of sizes 2k or less for network transfers.

No packets arrived out of order, nor were data lost even at the maximum rate. This agrees well with the high reliability observed when collecting and analyzing gigabytes of data in the field using CHESS modules.

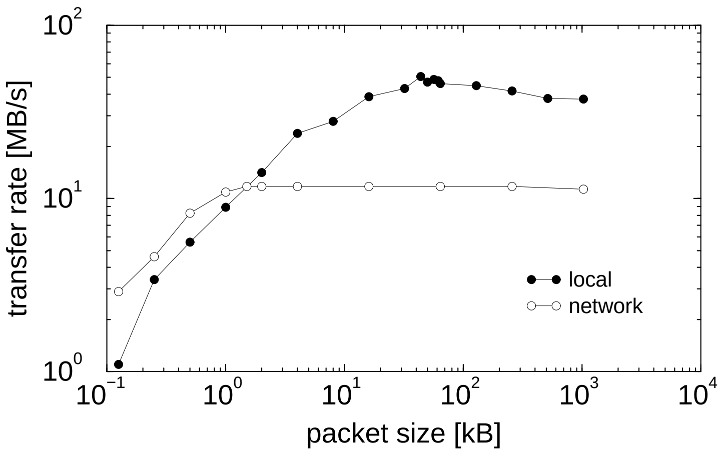


Figure 4.3 The raw data transfer rate between two CHESS modules running on a single computer (closed circles) and between two computers on a 100 Mbit/s network (open circles)

4.3 CHESS graphical tools

4.3.1 ChConnStatus

ChConnStatus displays information on the status of the data link to the server. The display is given as a single line. Rightmost, a “?” button is shown. Pressing this button brings up a page with details on the data received, such as sensor names, data type and so on. A circle which is green when the connection is up, and red when connection failed, is shown next to the button. Last, the server name and port number are shown.

Upon construction, a pointer to the monitored ChConn is provided. The ChConnChange constructor connects the ChConn::ConnChange(bool) signal to its slots, so that no active attention is needed from the programmer to update the display.

4.3.2 ChSendStatus

ChSendStatus displays information on a transmitting socket. The display states which port number data are transmitted on, and how many clients are connected. In the current CHESS standard, a transmitting ChSocket is present only when the module is connected data source. Thus, the programmer must provide a pointer to the transmitting ChTrans every time is it constructed, and a NULL pointer when no transmitting socket is present using the SetSock function.

4.3.3 ChCellLine and ChCellSelect

ChCellSelect provides a display where the user may select data cells. Each line is a ChCellLine object displaying a toggle button and the cell name. This interactive selection method is, so far, only used in the SimpleViewer module.

The ChCellSelect constructor is provided with a pointer to a boolean array. Each time a button is toggled, the state of the corresponding boolean variable is changed. ChCellSelect emits the FlChanged(int) signal when this event occurs, allowing the module to immediately update its state.

4.3.4 ChInfoScreen

ChInfoScreen provides a text screen with “close” and “clear” buttons. When the buttons are pressed, the finished() and Clear() signals are emitted, respectively. The host module may add text at the beginning or end of the display(AddStart and AddEnd), change the display auto-update status (SetAutoUpdate), actively redraw the text (DoRepaint), and change the font (SetFont).

4.3.5 ChLogSpinBox

ChLogSpinBox is derived from the Qt QSpinBox class, but provides a spinbox with exponentially increasing steps by doubling the value for each click upwards, and halving it for each click downwards.

4.4 CHESS file tools

4.4.1 ChFile

ChFile is a base class to be inherited by saving modules and provides the interface and graphical functionality for saving files. The user may select either binary or ASCII file formats, type in or use a browser to select the file name, and click the “Save” toggle button to start or stop saving.

The class also imposes some safety measures. The “Save” button may only be clicked when a non-existing filename is selected, and the file format may only be changed when the module is not in the saving mode.

The calling routine must overload the BinFileAppend, AscFileAppend and BinHeader functions to implement the data transfer details. The OnToggleSave method must be overloaded to implement save specific details.

The class is used as follows: The constructor is provided with the location of the file display on the host module GUI. After initialization, the paintButtons function is called to construct the actual button elements. Thereafter, the FileAppend() wrapper function should be called every time new data is available.

4.5 How to program a CHESS module

In order for the CHESS modules to be easily maintainable, they should use the functions provided by libCHESS as much as possible, and also adhere to the following guidelines:

- Do not rely on the graphical interface for module initialization. The modules are meant to be auto-started from scripts and should be completely configurable using command line options and configuration files.
- Use the ChTrans and ChConn classes for data transfer, not the base ChSocket class. This adds stability and flexibility to the system.
- Do not generally assume that the data has a given type. Signal processing functions may be implemented effectively using template classes and a wrapper function selecting the correct data types. This adds flexibility to the interconnection of modules.
- Destroy the transmitting socket whenever the data source is unavailable, and restore it when data becomes available again. This practice clearly identifies situations where data is unavailable, and is used by modules such as AutoRecord to close old and open new files.
- Utilize the signal/slot mechanism built into ChBuffer and its derived classes to control execution of the module. That is, trigger signal processing when the input buffer NewData signal is emitted, and emit the corresponding signal of the output buffer to transfer the data to the clients.

Consult the source code of the existing CHESS modules for further details and examples.

5 CHESS PIECES

The modules already written for CHESS are documented in the following chapter. The details of the signal processing algorithms are given, together with configuration file formats and command line options.

5.1 Conventions

The most common command line options follow the convention outlined in Table 5.1. Some modules use additional command line options, please see the documentation for the specific module.

	arg	comment
c	str	configuration file name
h	str	host name of data source
p	int	port number of data source
s	int	send socket
f	str	data file name

Table 5.1 Frequently used command line options

Comments in a configuration file must be preceded with a hash mark “#” at the beginning of the line.

The cell number of the first cell is 0.

5.2 Signal processing modules

5.2.1 Downsample

Downsample is a finite impulse response (FIR) filter and downsample module. A lowpass filter is normally used to avoid aliasing effects. The configuration file provides the module with the downsample rate (DSR), filter length and filter coefficients using the following format:

```
<downsample rate DSR>
<filter length>
<filter coefficient>
<filter coefficient>
....
```

The averaging filter is applied at every DSR’th data point. The result is pushed to the output buffer in the same format as the source data. Downsample may be used as a FIR filtering module by setting DSR to 1, and may be used for keeping every DSR’th data point by selecting a filter of length 1 and coefficient 1.

The filter introduces a delay equal to half the filter length. The timestamps of the resulting data points are corrected to reflect this delay.

5.2.2 Ellip

Ellip implements an infinite impulse response (IIR) filter with two coefficients in both the moving average part and the autoregressive part. Such filters are useful in cases when a FIR filter would be prohibitively long or introduce an unacceptable delay.

The configuration file has the format:

```
<AR coeff 0> <AR coeff 1>
<MA coeff 0> <MA coeff 1>
```

5.2.3 Normalize

Normalize is a strong normalization filter subtracting the average from the signal. The average is updated as long as the “Normalize” toggle button is in. This module is designed to be used when analyzing a fixed length signal, and not as a part of a permanent signal processing system.

5.3 Transformation modules

5.3.1 BendMoment

BendMoment implements the strain to global bending moment transformation developed by Alf Egil Jensen at FiReCo AS(6). The module takes strain data as input, and outputs bending moments as 4-byte floating point numbers. The sensors to be used in the transformation and the transformation matrices are provided in the configuration file.

```
NN NN NN NN NN <cell numbers used in transf matrix 1>

TT TT TT TT TT <transformation matrix 1>
TT TT TT TT TT
TT TT TT TT TT
TT TT TT TT TT
TT TT TT TT TT

NN NN NN NN NN <cell numbers used in transf matrix 2>

TT TT TT TT TT <transformation matrix 2>
TT TT TT TT TT
TT TT TT TT TT
TT TT TT TT TT
TT TT TT TT TT
```

Work is currently going on in the CHESS project to improve this transformation method. Using singular value decomposition and pseudo-inverse matrices, more sensors can be included in the estimates to give better results.

5.3.2 Raw2Strain

Storing raw rampcount values(3) is preferred when data is stored for later analysis. The Raw2Strain module converts the rampcounts to strain values using the calibration data entered in a FabryPerot data acquisition module configuration file. The data are output as floating point numbers to utilize the sub-microstrain resolution available in the dataset.

5.4 Data flow control

5.4.1 DataSwitch

DataSwitch is a module used to control the data flow to a signal processing network. It is used for turning off processing and logging when the ship is at shore, allowing the acquisition hardware and software to run continuously. The data flow is turned on using the “Run” button, allowing the crew to start data processing, display and logging at the press of a single button.

The data sources and corresponding send ports are given in the configuration file.

```
<server name> <server port> <send port>
<server name> <server port> <send port>
....
```

5.4.2 CellFilter

Cell filter is a module selecting a subset of cells from the input data stream. This may be useful for selecting a few interesting sensors for logging to disk, thereby reducing file size, or sorting the sequence of the cell.

The sequence of the cells to be transmitted is given in the configuration file.

```
<Cell #1>
<Cell #2>
...
```

5.5 File recording

5.5.1 Recorder

The Recorder module logs a file to disk either in a ASCII or binary format. The binary format first saves a header with full info on the data stream, as detailed in chapter 4.2.1, and then logs the received data to disk. The binary data may be played back at a later stage using the Playback module.

The ASCII mode is useful for converting the binary format to an ASCII format readable by other software.

This module is intended for interactive use, and inherits the ChFile class described in chapter 4.4.1.

5.5.2 AutoRecord

AutoRecord is an automated data logger using the same binary format as the Record module. A base filename is provided at the command line. Upon connection to the data server, the module generates a unique filename by adding the current UTC time and a “.bin” extension to the base filename. The file is closed when the connection to the server goes down.

The ChConn input socket will repeatedly attempt reconnection, and a new file is generated when the connection attempt succeeds. Thus, this module is useful as a data logger in a permanent surveillance application.

5.5.3 EventLog

EventLog monitors the absolute value of the input data. When it is above a trigger level set in the configuration file, the UTC timestamp is appended to a log file. This simplifies the search for extreme events during post-processing of the data.

EventLog takes a number of command line parameters controlling its behavior, see table 5.2

	arg		comment
c	str	Opt	configuration file name
h	str	Req	host name of data source
p	int	Req	port number of data source
f	str	Req	log file name
n	int	Req	minimum distance between different events

Table 5.2 EventLog command line options

If no configuration file is provided, the default trigger level of 700 is used for all data.

5.5.4 MaxLog

MaxLog records the maxima and minima of the input data to a binary file together with the timestamp. The extremal points are identified from a changing sign in the difference between neighboring data points. This application uses a special binary data format, and the data may only be converted to ASCII format using the ConvMaxLog program described below.

Experience with this module indicates that data should be as smooth as possible. It may therefore be beneficial to convert strain data from short to floating point format early in the processing sequence, as shown in Figure 2.2.

5.6 Display modules

5.6.1 SimpleViewer

SimpleViewer is, as the name implies, a simple datastream viewing module. The user may select the data cells to view by selecting the corresponding cell names. The graph is updated each time a new data packet is received, and the latest 512 data points are shown. Thus, if packets are longer than 512 points, some data will never be displayed.

The viewer has no autoscale function, but the y-axis scale may be doubled or halved using the “+” and “-” buttons. The window may also be resized.

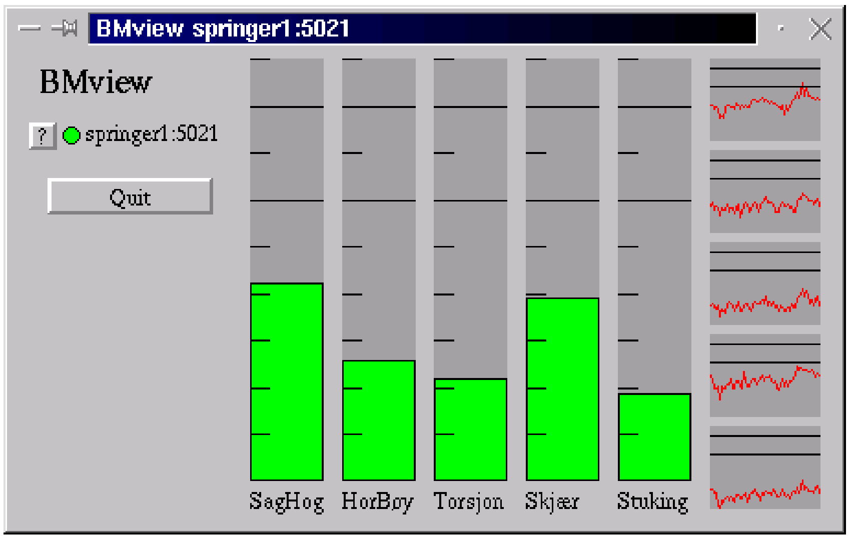


Figure 5.1 The BMview display

5.6.2 BMview

BMview is a bending moment viewing module designed after discussions with Skjold's captain. The bargraph shows the maximum bending moments in the last data packet, while the graphs show a history of maximum moments (see Figure 5.1).

The scales are logarithmic, and each line represents a doubling of the load. The solid lines represent 25% and 100% of the design limit given in the configuration file.

The configuration file also defines the number of packets to be used per point in the history plots. This number thus controls the length of the history. The configuration file format is given below.

```
<Number of packets per plot point>
<limit 1> <name 1>
<limit 2> <name 2>
....
```

5.6.3 MaxMin

MaxMin is a module for displaying the maximum and minimum value of the data for each sensor. The results are displayed in a ChInfoScreen window when the "Display" button[1] is pressed. The values may be reset using the "Reset" button.

[1]CHECK

The module is useful when the maximum amplitude of the signal is sought after, for instance during cannon firing tests.

5.6.4 LaserAlarm

LaserAlarm is a module that detects sensor fallout. It was necessary to include this monitoring facility for the long-term system tests during winter 2000 to alert the crew in cases of system errors, such as the light source being switched off, or malfunctions of the automated error correction facilities in the FabryPerot module.

The module calculates the standard deviation of the data in a data packet. The alarm is triggered when the SD is 0 for any sensor. Alarms are displayed as red flashing background in the LaserAlarm window, and with the system bell sound.

5.7 Sources and converters

5.7.1 FabryPerot

The FabryPerot data acquisition module is described in detail in a separate report(3). This module is the main data source in the system, and communicates with a realtime operating system module (rt_epp_handler) to interface with the data collection hardware. The acquisition module has logging, data correction, status indication and socket communication facilities. Modules may connect to two ports for data transfer. On the lower port number, typically 5001, strain data are transmitted, while raw data are transmitted on the higher port number (5002). It may also be used as a stand alone module, logging data to disk in binary or ASCII formats.

5.7.2 Player

The Player is a FabryPerot module with a file data source. It contains the same correction, display and communication facilities as FabryPerot, in addition to a window for playback control(3). This module also allows correction routines to be applied during post-processing of the data.

5.7.3 Playback

Playback is a module for transmitting a datastream saved using Record or AutoRecord to the network. The user selects the source file and playback speed, and may start playback once the file is loaded. The playback speed is controlled by changing the size of the data packets transferred, while transmitting at a fixed interval of XX ms[2]. When transmitting at the maximum speed, the maximum packet size is used and the transmit interval is set to 0.

5.7.4 ConvMaxLog

The ConvMaxLog program converts data saved in the special MaxLog data format to an ASCII file. After loading the binary file, the user may select a channel to extract data from. The file conversion is started by pressing the “convert” button.

6 FUTURE MODULES

This chapter points to some ideas for the future development of modules for the CHESS system. Developing modules implementing some sort of transformation on the data is usually done in a few hours since all modules build on libCHESS and use the same functionality. Typically, the programmer only has to change details in the user interface, details in local variables necessary for the transform, and implement the transform algorithm itself.

The modules outlined below represent a greater challenge since they introduce other data sources, heavy signal processing algorithms or graphics/presentation challenges.

6.1 New data sources

6.1.1 NMEA

This should be a general module for accepting and decoding NMEA-0183 messages(7). Many marine instruments, such as GPS receivers, transmit such messages on the serial port. The messages are ASCII text strings defined by the NMEA protocol.

6.1.2 Integration with ship control systems

The possibility of exchanging data with the ship control systems should be explored. This could be a two-way data exchange procedure, where the CHESS signal processing network receives data on speed, course, trim settings, cushion pressure and other parameters that could be useful in evaluating the performance and predicting the probability of damage.

On the other hand, the ship systems may receive data from the hull surveillance module to

[2]Check interval

display information and alarms on the bridge and in the machine control room.

As the control systems communicate on a TCP/IP network there is no fundamental barrier for data exchange to take place.

6.1.3 DAQ-card integration

Several measurement systems give voltage-level signals that can be digitized by a data acquisition card. Such signals include outputs from amplifiers for accelerometers, conventional strain gauges and pressure sensors, as well signals from high-speed interferometric sensor techniques.

The need to measure such signals are present in most data acquisition missions, and should be integrated in the CHESS network.

Drivers for Linux for some DAQ cards are available form the Comedi project(8), but some of these drivers carry a performance penalty compared to the Windows versions since DMA transfers are not implemented.

A possible solution would therefore be to run the DAQ acquisition computer on the NT operating system, and transfer the data to the CHESS network on a TCP/IP socket.

6.2 Signal processing algorithms

6.2.1 Align

So far, all data that have been processed in the CHESS network have been acquired using a common time base. A challenge in data management is introduced by integrating more sources into the system.

There are several possible solutions to this problem. First, data may be up– or downsampled to be placed on a measurement grid with a common timebase. Alternatively, one might use a more complex structure with a separate data path for each measurement type. The strategy for the further development of the structure of CHESS data streams should therefore be carefully considered.

6.2.2 Time-frequency analysis

So far, all time-frequency analysis of CHESS data has been carried out in the post-processing stage by importing data into matlab. The future signal processing system may include wavelet decomposition or similar methods. Frequency decomposition may be useful in classifying events, filtering and in a possible damage detection module.

6.3 Display applications

Display modules should be optimized for the intended audience. A display module presenting information to the ship's officers should only give the information necessary for optimal operation of the ship. Display modules intended for analysis, research and development may have a much larger complexity.

As of now, BMview is the only module designed following discussions with the captain. To further improve the quality of the CHESS operator interfaces, CHESS may draw on resources from experts in the field, such as the IFE reactor project in Halden, Marintek and various commercial suppliers.

References

(1) Pran K, Sagvolden G, Farsund Ø, Havsgård G B, Wang G (2000): A summary of project 711 ”Fiberoptisk skrogovervåkning” (CHESS), FFI/RAPPORT-2000/01298, Forsvarets forskningsinstitutt (Approved for public release. Distribution unlimited).

(2) Pran K, Sagvolden G, Farsund Ø, Havsgård G B, Wang G (2000): Preliminary design proposal for a fibre optic structure monitoring system for surface effect ships, FFI/NOTAT-2000/01297, Forsvarets forskningsinstitutt (Approved for public release. Distribution unlimited).

(3) Sagvolden G (2000): Data acquisition software for Fabry-Perot based fiber Bragg grating interrogation hardware, FFI/RAPPORT-2000/01293, Forsvarets forskningsinstitutt (Approved for public release. Distribution unlimited).

(4) Sagvolden G, Farsund Ø, Lundberg Ø, Pran K (2000): Displacement measurements during firing tests on KNM Skjold October 1999, FFI/NOTAT-2000/01295, Forsvarets forskningsinstitutt (Distribution limited).

(5) Http://www.troll.no/.

(6) Jensen A E, Taby J, Pran K, Sagvolden G, Wang G: Measurement of global loads on a full scale SES vessel based on strain measurements using networks of fibre optic Bragg sensors and extensive finite element analyses, submitted to J. of ship research.

(7) Http://www.nmea.org/.

(8) Http://stm.lbl.gov/comedi.

DISTRIBUTION LIST

FFIE **Dato:** 12 july 2000

RAPPORTTYPE (KRYSS AV) [✓ X] RAPP ☐ NOTAT ☐ RR	RAPPORT NR. 2000/01294	REFERANSE FFIE/711/116	RAPPORTENS DATO 12 july 2000
RAPPORTENS BESKYTTELSESGRAD UNCLASSIFIED		ANTALL EKS UTSTEDT 41	ANTALL SIDER 37
RAPPORTENS TITTEL MODULAR DISTRIBUTED SIGNAL PROCESSING NETWORK FOR CHESS		FORFATTER(E) SAGVOLDEN Geir	
FORDELING GODKJENT AV FORSKNINGSSJEF:		FORDELING GODKJENT AV AVDELINGSSJEF:	

EKSTERN FORDELING

ANTALL	EKS NR	TIL
		Naval Research Laboratory
1		Fiber Optic Smart Structures Section
1		V/Gregg Johnson
1		V/ Sandeep Vohra
1		V/ Gary Cogdell
		Code 5600
		Washington DC 20375
		USA
1		SFK, Teknisk Avdeling
1		V/ Steinar Nilsen
1		V/ Atle Sannes
		Postboks 3, Haakonsvern
		5086 BERGEN

INTERN FORDELING

ANTALL	EKS NR	TIL
14		FFI-Bibl
1		Adm direktør/stabssjef
1		FFIE
1		FFISYS
1		FFIBM
1		FFIN
1		Gunnar Wang, FFIE
1		Karianne Pran, FFIE
1		Øystein Farsund, FFIE
1		Geir Sagvolden, FFIE
10		Arkiv, FFIE
1		FFI-veven

Retningslinjer for fordeling og forsendelse er gitt i Oraklet, Bind I, Bestemmelser om publikasjoner for Forsvarets forskningsinstitutt, pkt 2 og 5. Benytt ny side om nødvendig.