Project Outline

Project 'BCI2000'

Gerwin Schalk
Dennis J. McFarland
Jürgen Mellinger

New York State Department of Health

Wadsworth Center
Brain-Computer Interface Research and Development Program

Eberhard–Karls–University Tübingen

Institute of Medical Psychology and Behavioral Neurobiology

Sponsors
Jonathan R. Wolpaw and Niels Birbaumer

Albany, NY
February 2000–July 2004

Contents

Chapter 1

Introduction

1.1 Involved Groups

This document describes a project whose core contributors are the Brain-Computer Interface Research and Development Program at the Wadsworth Center of the New York State Department of Health and the Institute of Medical Psychology and Behavioral Neurobiology at the Eberhard-Karls-University in Tübingen, Germany.

1.2 Project Sponsors and Management

The BCI2000 project is sponsored by Jonathan R. Wolpaw, MD and Prof. Dr. phil. Niels Birbaumer. The project is managed by Gerwin Schalk, MS.

1.3 Acknowledgement

We want to thank Dr. Thilo Hinterberger for his contributions during the initial phase of the project and Dr. Jouri Perelmouter for his input during the project design phase.

Chapter 2

Context and Purpose

In recent years, many laboratories have begun to develop brain-computer interface (BCI) systems that provide communication and control capabilities to people with severe motor disabilities. Further progress and realization of practical applications depends on systematic evaluations and comparisons of different brain signals, recording methods, processing algorithms, output formats, and operating protocols. However, the typical BCI system is designed specifically for one particular BCI method, and is therefore not suited to the systematic studies that are essential for continued progress. Furthermore, BCI design depends on the mastery of diverse areas such as computer science, neuroscience, physiology and psychology, and thus a tool that could facilitate the interaction and integration of competency in these areas would greatly foster the future development of BCI technology.

In response to this problem, we have developed and tested a general-purpose BCI research and development platform, called BCI2000. BCI2000 can incorporate alone or in combination any brain signals, signal processing methods, output devices, and operating protocols. To date, more than 25 laboratories around the world are using BCI2000 in a variety of studies. The BCI2000 system is available with free of charge for research or educational purposes at `http://www.bci2000.org`.

The background and rationale, as well as the general concept of BCI2000 is described in detail in a paper in *IEEE Trans Biomed Eng* (Schalk et al., 2004). This document goes beyond the scope of this article in that it defines the technical implementation-independent structures of BCI2000 (such as the communication protocol and the file format), i.e., in other words, the *BCI2000 Standard*. This information is sufficient to create any addition to BCI2000 (in any language and on any operating system) that is compatible with this standard. BCI2000 is currently implemented using Borland C++ Builder on Windows platforms. For specific questions about these implementations (such as how to implement your own signal processing routines), please refer to the *BCI2000 Software Design Document*.

Chapter 3

System Design

3.1 Overview

BCI2000 consists of four modules shown in Figure 3.1 that communicate with each other. These modules are called *Source*, *Signal Processing*, *Application*, and *Operator*. We will refer to the three modules Source, Signal Processing and Application together as *core modules*.

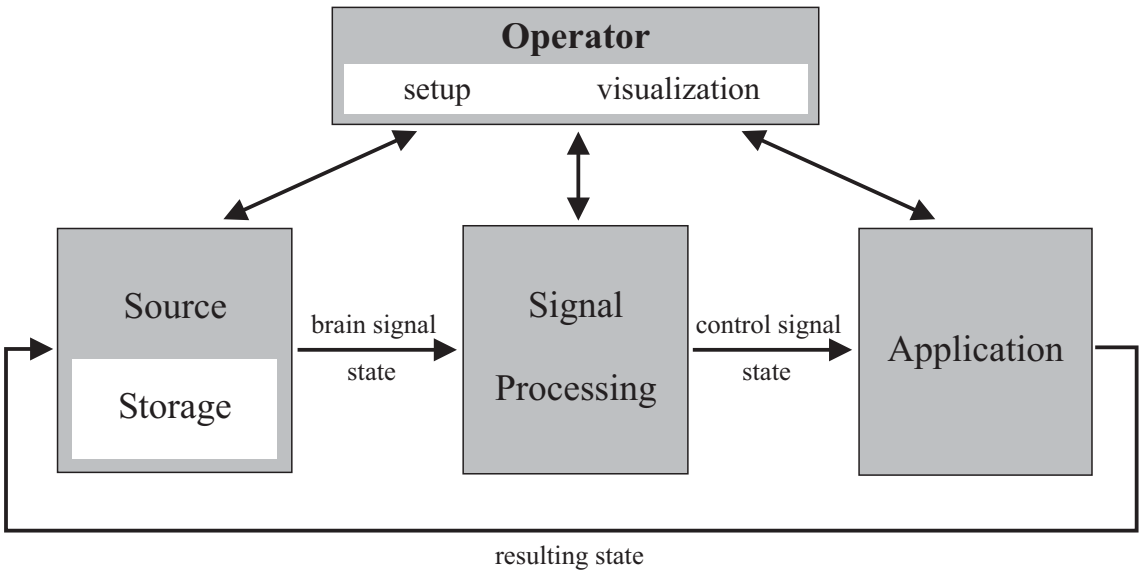

Figure 3.1: The functional modules and their interfaces

3.2 Core–Operator Communication

3.2.1 Introduction

The same communication protocol is used between each of the three core modules and the operator module. Although it can be implemented on top of any transport protocol, it assumes the reliability of TCP (i.e., the protocol does not support acknowledgements or packet sorting or any other means of error correction).

Any message sent to or from any module is wrapped into packets as described in section 3.2.2. This simple protocol allows for the delivery of data (e.g., parameters, visualization data, etc.), even if its content and nature are not known *a priori*.

3.2.2 Protocol Definition

Communication consists of messages that are sent and received asynchronously between any core module and the operator module. Each message starts with a one-byte content descriptor and one-byte descriptor supplement, followed by a number that describes the length of the content.

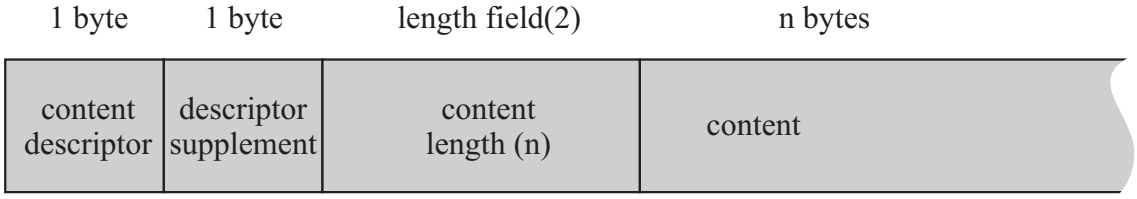

Figure 3.2: Layout of one message in the protocol

The element denoted by "length field(2)" was originally a two-byte integer field for the content length in little endian format. To allow for messages longer than 64k, we introduced a backwards-compatible extension: if the length is below 65535, it will still be transmitted as a two-byte integer in little endian format. Otherwise, the two bytes will contain the value 65535, and be followed by a decimal ASCII representation of the length, terminated with a zero byte. For other one- and two-byte length fields occurring in the protocol, the same scheme applies, generalized to be a "length field(original number of bytes)" (Figure 3.3).

3.2.3 Descriptor=1: Status Information Format

This section describes the format of the message when the core message is a status information string (i.e., the message's descriptor is 1). In this case, the aforementioned content data is a line of ASCII characters in the following format:

```
xxx: status-line-text
```

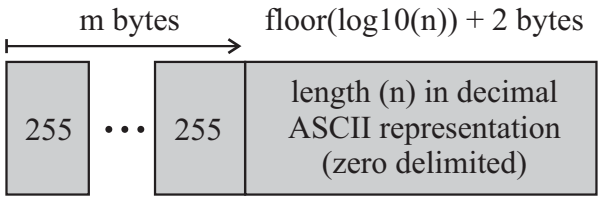

Figure 3.3: Detailed layout of a length field(m) for a length $n \geq 2^m - 1$

descriptor	**description**
1	status information string
2	system parameter
3	system state
4	visualization data
5	state vector
6	system command

Table 3.1: The currently defined content descriptors

xxx is a three digit number that describes the content of the status information string. The first digit is '1' for status information, '2' for successful operation, '3' for recoverable errors and '4' for fatal errors. The two remaining digits define the exact nature of the message, followed by a plain description.

This procedure is used to communicate errors and to convey status information (e.g., the operator module may display the remaining disc space on the Source module's machine.)

3.2.4 Descriptor=2: Parameter Format

This section describes the format of the message when the core message is of type system parameter (i.e., the message's descriptor is 2). In this case, content data is a line of ASCII characters in the following format (Table 3.3 summarizes the individual components in the parameter string):

```
Section DataType Name= Value DefaultValue LowRange HighRange
    // Comment CRLF
```

If DataType is *list, intlist* or *floatlist*, the format is as follows:

```
Section DataType Name= (NumValues|Labels) Value(s) DefaultValue
    LowRange HighRange // Comment CRLF
```

where `Labels` is a list of textual labels that optionally substitutes the number denoted by `NumValues`. A list of labels is enclosed by `{}` or `[]`, as in `[low medium high]`.

If DataType is *matrix*, the format is as follows:

```
Section DataType Name= (NumValuesDim1|LabelsDim1)
    (NumValuesDim2|LabelsDim2) Value(s) DefaultValue LowRange
    HighRange // Comment CRLF
```

where again each NumValues entry may be substituted by a list of textual labels.

Special characters To allow for special characters and white space within parameter values and textual labels, an URL-like encoding scheme is adopted. In this encoding, a `%` character followed by up to two hexadecimal digits represents a byte value in hexadecimal notation which is interpreted according to the ASCII-Latin1 character table. Thus, `%20` represents a space character, and `%`, `%0`, and `%00` all represent an empty string value; `%%` represents the `%` character itself. The line

```
Demo string SomeString= a%20string%20with%20spaces % % %
    // White space example
```

defines a parameter *SomeString* which contains the value *"a string with spaces"*. The single `%` characters indicate that the *DefaultValue*, *LowRange* and *HighRange* fields should be empty strings.

Sub-parameters Any parameter value may itself be a sub-parameter. Sub-parameters are represented by a short-form matrix definition omitting the *Section* and *Name* fields, enclosed in a pair of braces:

```
Demo matrix NestedMatrices= 1 2 11 { matrix 2 2 1211 1212 1221 1222 }
    // Nested matrix example
```

will define a matrix parameter whose 1,2 entry is a 2x2 matrix. While the syntax allows for any combination of parameter and subparameter types, the current implementation of the parameter editor GUI does only support matrix-type subparameters within matrices as in the example above.

Display Format The comment line that is introduced by the two slashes (//) may contain a format identifier that is used by the Operator module to modify the display of the particular parameter. Format identifiers are strictly optional and are introduced as follows: `(identifier)`. Currently, the following format identifiers are implemented:

`(enumeration)` The parameter value is presented as a drop down menu, with entries corresponding to the possible values listed in the comment. The first part of the comment appears above the drop down menu. All interpunction characters present in the comment are ignored.

Identifier	Description
`(enumeration)`	a choice from an enumerated set of values
`(boolean)`	a yes/no choice
`(inputfile)`	path to a file to be opened for reading
`(outputfile)`	path to a file to be opened for writing
`(directory)`	path to a directory
`(color)`	RGB color

Table 3.2: The currently defined format identifiers

The parameter's data type must be `int`. All possible values must appear in the comment, and the parameter's LowRange and HighRange fields must be consistent with the enumeration. LowRange will usually be 0, but may be any integer.

```
Breakfast int BreakfastDrink= 1 1 1 3
  // Drink for breakfast: 1 Tea, 2 Coffee, 3 Juice (enumeration)
```

will display a drop down menu with entries "Tea," "Coffee,", and "Juice." The menu will be labeled "Drink for breakfast."

`(boolean)` The parameter value is presented as a check box; the first part of the comment appears to the right of the check box. LowRange and HighRange must be 0 and 1.

The parameter's data type must be `int`. To ensure human readability of parameter files, the possible values and their meaning may appear in the comment (e. g. `0: no, 1: yes`) but will not be displayed with the check box.

```
Breakfast int ServeBreakfast= 1 1 0 1
  // Serve breakfast: 0 no, 1 yes (boolean)
```

will display a check box labeled "Serve breakfast."

`(inputfile)(outputfile)(directory)` The parameter value is presented as an edit field. Beside the edit field, a button exists that opens up a file or directory choosing dialog when clicked.

The parameter's data type must be `string`.

```
Breakfast string WakeupSound= doorbell.wav % % %
  // Sound to play in the morning (inputfile)
```

`(color)` The parameter value is presented as an edit field. Right to the edit field, a button exists that opens up a color chooser dialog when clicked.

The parameter's data type must be `string`, with its value containing the color in hexadecimal RGB encoding:

```
Breakfast string TableClothColor= 0x00FF00 0xFFFFFF 0x000000 0xFFFFFF
  // Color of table cloth to put up for breakfast (color)
```

Grouping Parameters A user interface may use parameters' `Section` fields to collect parameters into groups, e.g. by displaying all parameters with identical section fields on the same register tab of a GUI parameter editor dialog window.

In the `Section` field, finer grained grouping may be expressed by specifying subsections separated by colon characters, e.g. a `Section` value of `UsrTask:WindowDimensions` will indicate that a parameter belongs to a `WindowDimensions` subsection of a section called `UsrTask`. A parameter editor implementation might choose to display the respective parameter on a register tab called "UsrTask" and inside a group box labelled "WindowDimensions".

Although any number of sub-sections may be present in the `Section` field, a user interface implementation may ignore sub-section entries below a level chosen by the implementer.

The parameter format does not only apply to the interaction between core and operator, but also reflects the data format of BCI2000 parameter files (i.e., files with extension .prm) as well as the parameter portion of the BCI2000 data files (i.e., files with extension .dat) files. The core modules and the operator module also use this format to communicate in the system initialization phase (as described in section 3.6).

Section
Type
Name
[(NumValuesDim1|{ LabelDim1.1 LabelDim1.2 ... }) [(NumValuesDim2|{ LabelDim2.1 LabelDim2.2 ... })]]
Value(s)
DefaultValue
LowRange
HighRange
Comment incl. format identifiers

Table 3.3: Definition of variables in the parameter string

For data types *list, intlist* and *floatlist*, *NumValues* indicates how many values are following. The separate values are then delimited by white spaces. In case of data type *matrix*, the transmitted values represent the matrix as follows: the first NumValuesDim2 values are value(0, t), followed by NumValuesDim2 values value(1,

t), etc.; in case that labels are specified in place of value counts, the analogy holds with the value counts given by the number of labels for the dimension in question.

Each core module knows the data types of the parameters it is requesting and thus does not need a parameter data type definition. However, the operator module uses the data type definitions as listed in table 3.3 to create their graphical representation. Section 3.4 describes a list of pre-defined system parameters.

3.2.5 Descriptor=3: State Format

This section describes the format of the message when the core message is of type system state (i.e., the message's descriptor is 3). In this case, the aforementioned content data is a line of ASCII characters (parameters delimited by spaces) in the following format:

```
Name Length Value ByteLocation BitLocation CRLF
```

with BitLocation ranging from 0..7. The core modules and the operator module use this format to communicate in the system initialization phase (as described in section 3.6), as well as during system performance (section 3.7) and for system termination (section 3.9). Table 3.4 defines the maximum length for each parameter.

Name	**Max. Length**
Name	30
Length	2
Value	5
ByteLocation	1
BitLocation	1

Table 3.4: Definition for variables in the state string

3.2.6 Descriptor=4: Visualization Data Format

This section describes the format of the message when the core message is of type visualization data (i.e., the message's descriptor is 4). In this case (see figure 3.4), the content descriptor describes the requested visualization type. The only currently defined types are 1 (a graph of n channels and m samples), 2 (a text memo), and 255 (visualization configuration).

Figure 3.5 illustrates the protocol when the visualization type is 1. The source identifier defines a unique number identifying the process/filter that generated the data. The data type can be

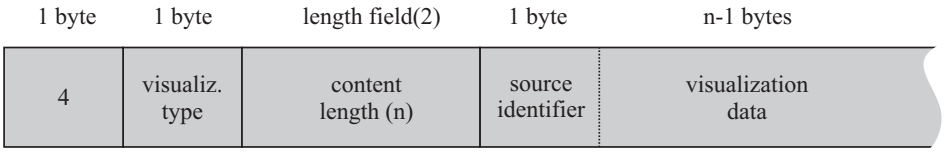

Figure 3.4: One message in the protocol if it is of type "visualization data"

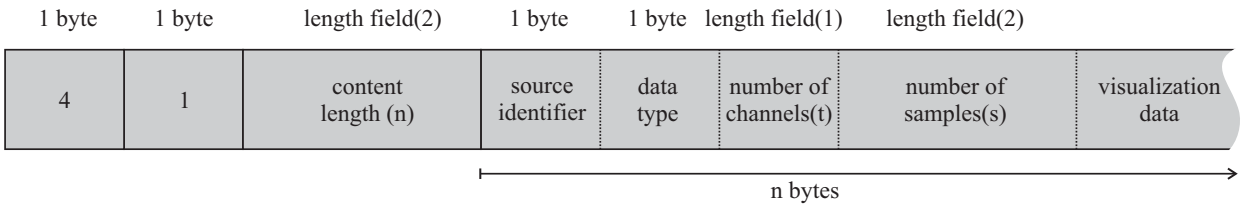

Figure 3.5: One message if visualization type is 1 (i.e., a graph)

- 0 (`SignalType::int16`) for integers in little endian format.
- 1 (`SignalType::float24`) for 3-byte floating-point values: The first two bytes (i.e., A) define the mantissa (signed two-byte integers in little endian format) and the third byte (i.e., B) defines the exponent (signed one-byte integer). The actual floating point value is then calculated as follows: $value = A * 10^B$.
- 2 (`SignalType::float32`) for 4-byte floating-point values in IEEE 754 format transmitted in little endian byte order.
- 3 (`SignalType::int32`) for 4-byte signed integer values transmitted in little endian byte order.

The number of channels and samples are self explanatory. Figure 3.6 illustrates how the data is transferred.

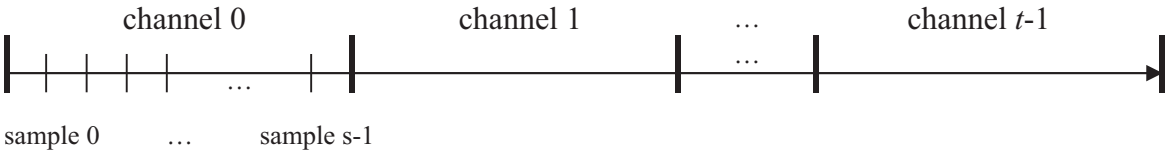

Figure 3.6: Graphical representation of the transmitted visualization data format

Figure 3.7 illustrates the protocol when the visualization type is 2. The source identifier is a number uniquely identifying the process/filter that generated the data. The following ASCII text is zero delimited.

Figure 3.8 illustrates the protocol when the visualization type is 255. The source identifier is a number identifying the process/filter that generated the data. The different configuration IDs are described in Table 3.5.

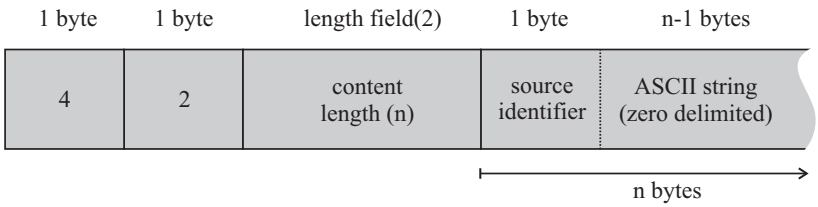

Figure 3.7: One message if visualization type is 2 (i.e., a text memo)

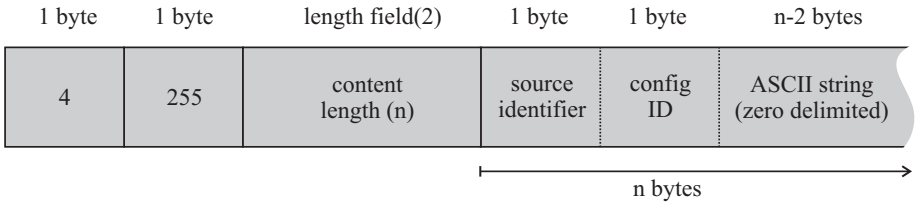

Figure 3.8: One message if visualization type is 255 (i.e., visualization configuration)

The ASCII string then contains the configuration option, as defined by the configuration ID. For example, it might contain "128" if the configuration ID is 4. This will configure the graph to contain exactly 128 samples. When the configuration ID is 5 or 6 (axis labels), the ASCII string consists of a sample number (three digits), a space, and the axis label. Thus, one message configures exactly one axis label. As an example and for an X-axis label, the string "003 4.75 Hz" would result in a graph, in which the third sample is labeled "4.75 Hz."

3.2.7 Descriptor=5: State Vector

This section describes the format of the message when the core message is of type state vector (i.e., the message's descriptor is 5).

The state vector is defined by a series of *StateVectorLength* subsequent bytes. The structure of these bytes is defined in 3.3.3.

3.2.8 Descriptor=6: System Command

This section describes the format of the message when the core message is of type system command (i.e., the message's descriptor is 6).

The system command consists of an ASCII string that may end with a zero byte (i.e., ASCII code 0).

The nature of these system commands is defined by the specific implementation of the modules.

Cfg ID	Description
1	Window Title
2	Minimum Value
3	Maximum Value
4	Number of Samples
5	X Axis Label
6	Y Axis Label
7	Number of Subsequent Channels displayed on the same base line
8	Graph Type (Polyline or 2D Field)
9	Polyline Graph Type
10	Show Base Lines (1 or 0)
11	List of Channel Colors
12	2D Field Graph Type
13	Unit for Sample dimension
14	Unit for Channel dimension
15	Unit for Values
16	Number of Lines for a Memo

Table 3.5: The various IDs for visualization configuration

3.3 Inter–Core Communication

3.3.1 Introduction

Unlike the bidirectional communication between core modules and the operator module, communication within the core modules is unidirectional. The initial setup determines the exact nature of this communication and data is transmitted with the same protocol as described in Section 3.2.2. The following sections describe the transferred brain signal, state vector, and control signal. The state vector is always transmitted before the brain signal or control signal. Whenever *Running* switches from 0 to 1, the system is re-initialized.

3.3.2 Brain Signal Format

The brain signal is transmitted similarly to visualization data (i.e., as described in Section 3.2.6). The visualization type is set to 1 (i.e., graph), source identifier is set to 0. Data type, channels and samples reflect the actual format of data transmitted.

3.3.3 State Vector Format

The state vector is transferred as type state vector (see Section 3.2.7). The content descriptor is 5 and the descriptor supplement is undefined. Its content consists of a

series of *StateVectorLength* bytes. The value of a given state within the state vector is determined by its byte/bit location and length definition. The bits in the state vector are always sorted in ascending order, e.g., for a state with a length of 7 bits, starting at byte location 2, bit location 3, bit zero is first (byte 2, bit 3), and the highest bit (bit 7) is last (byte 3, bit 1).

3.3.4 Control Signal Format

Control signals are transmitted similarly to the Brain Signal (see Section 3.3.2) with *NumControlSignals* channels and one sample per channel.

3.4 System-Wide Parameters

3.4.1 Introduction

System–wide parameters are published and initialized during system–startup, as described in section 3.6. Each parameter is described by the section it belongs to, a data type descriptor, its name, value(s), default value, and a range of valid numbers. Parameter names are unique across the entire system.

3.4.2 Defined Data Types

Data Type	**Description**
char	one character
string	a string of characters
int	a 16-bit signed integer
longint	a 32-bit signed integer
float	a single-precision floating point
bool	a boolean value (0..false, 1..true)
list	a list of untyped values
intlist	a list of 16-bit signed integers
floatlist	a list of single-precision floating points
matrix	a matrix of untyped values

3.4.3 Defined Sections and Parameter Names

Section names can be defined by the parameter definition. The only reserved section name is *System*. User-defined parameters may not go into this section. The BCI2000 standard requires the definition of the following parameters:

Section "Source"

Type	**Parameter Name**	**Description**
int	SoftwareCh	number of digitized and stored channels
int	SampleBlockSize	number of transmitted samples
intlist	TransmitChList	list of channels to transmit
int	SamplingRate	data acquisition's sampling rate in Hz

Section "Storage"

Type	**Parameter Name**	**Description**
string	SubjectName	subject alias
string	SubjectSession	session number (max. 3 characters)
string	SubjectRun	digit run number (max. 3 characters)
string	FileInitials	top level directory for saved files

Section "Filtering"

Type	Parameter Name	Description
int	NumControlSignals	number of transmitted control signals
int	AlignChannels	whether or not to align channels in time
floatlist	SourceChOffset	offset in A/D units
floatlist	SourceChGain	factor to convert A/D units to μV
floatlist	SourceChTimeOffset	offset of sample at each channel in time (from 0..1)

Section "System"

Type	Parameter Name	Description
string	EEGsourceIP	IP address the Source module listens on
int	EEGsourcePort	port the Source module listens on
string	SignalProcessingIP	IP address Signal Processing listens on
int	SignalProcessingPort	port Signal Processing listens on
string	ApplicationIP	IP address the Application module listens on
int	ApplicationPort	port the Application module listens on
int	StateVectorLength	the length of the state vector in bytes

3.5 System-Wide States

3.5.1 Introduction

While the purpose of system-wide parameters is to define the static configuration, the state information provides dynamic information about the current state of the system, e.g., whether the system is running or not.

States are published and initialized during system-startup, as described in section 3.6. A state is described by its name, its length (in bits) and its location (defined by start byte and bit in the state vector). The state vector is always transferred as a full number of bytes (specifically, *StateVectorLength* bytes) (i.e., the end is padded with zeroed bits if necessary).

3.5.2 The State Vector

While the operator module constructs the state vector after receiving all requests for states from the core modules, some states do not need to be to be requested – they are created automatically. They are:

Length	Name	Description
16	SourceTime	16-bit unsigned integer; resolution 1 ms
16	StimulusTime	16-bit unsigned integer; resolution 1 ms
1	Running	1: system is running, 0: system is suspended

3.6 System Initialization

3.6.1 Introduction

This section describes the system initialization process. Since the system is a distributed system of encapsulated modules, this procedure ensures a proper and well defined information flow at start-up.

3.6.2 Startup Sequence

The operator module must be started first. Since in most cases the IP address of the operator module can be more easily statically defined, its IP address and port number(s) have to be provided to the core modules. It listens on ports 4000 (for Source), 4001 (for Signal Processing), and 4002 (for Application) and waits for the respective core module to connect. Each can connect to its assigned port on the operator module in any order. Upon start–up, each core module opens a listening socket on an arbitrary port number.

3.6.3 Publishing Phase

Upon connection, each core module publishes its parameters to the operator module, as described in section 3.2.4. After publishing its parameters, each core module publishes the states it requests, as described in chapter 3.2.5. At this time, the operator module ignores every field except *Name* and *Length*, which it needs to construct the state vector.

For this and all subsequent communication, the modules use the protocol described in chapter 3.2.2. Following the last state, each core module sends a system command containing the string `EndOfState`. On receiving this command from all core modules, the operator module ends this initial publishing phase.

3.6.4 Information Phase

The operator module processes the received parameters and states. It creates a list of all parameters and all states and creates the state vector (double parameters or states are ignored). At this point, the operator module may modify the value of the parameters and states (depending on the investigator's input or the parameter file). The operator module then uses the same channel on which it received data from the core modules to send back to all core modules a list of all system-wide parameters and system-wide states (in any order). Since the IP address and port number on which the core modules listen for data from other core modules are published in system parameters as described, each module now knows where to send its data.

The connections from the core modules to the operator module remain open (all subsequent traffic will go through these connections).

As in the publishing phase, the Information Phase ends when a system command `EndOfState` is sent.

In order to maintain integrity throughout operation, no parameters or states should be added to or removed from the system beyond this point.

3.6.5 Preflight Phase

Each core module declares whether it can process data with the received parameters and states, or indicates errors by sending descriptions into an error channel. If any errors are indicated during the preflight phase, the module will not initiate the initialization phase; the operator module will display the errors, prompting the user to fix the problems detected, and not offer the "Start" option.

3.6.6 Initialization Phase

Each core module uses the information in the received parameters to configure and initialize its operation. It also opens an active (i.e., client) connection to the other core module it must connect to, i.e., Source opens a connection to Signal Processing, Signal Processing to Application and Application to Source.

Each core module sends a status message (see 3.2.3) to the operator that indicates either successful or failed initialization. The Initialization Phase ends when all core modules indicate successful configuration.

3.7 System is Running

At the end of the Initialization Phase, the system is fully configured. All parameters and states (and positions thereof in the state vector) are defined. During system operation, the Operator module must send states only to the Source module and Signal Processing and Application must disregard any state that the Operator does send to them.

The system is started when the Operator module sets the state *Running* to 1 and sends it to the Source module.

3.8 System is Suspended

The system is suspended when *Running* is 0. Any module shall disregard a change in parameters if the system is not suspended. Data flows through the system and it is up to each module to decide how to process these data. The Application, for

example, might give visual feedback that indicates that the system is suspended. As long as operation is suspended (i.e., *Running* is 0), any module might update system parameters and send them back to the Operator.

3.9 System Termination

If any module detects a dropped connection to the Operator, it must shut down all other socket connections and terminate.

3.10 File Formats

3.10.1 Data File

The data file consists of a header and the actual raw brain signals. The header consists of a definition of all system parameters and states. Thus, parameters must not change within one data file.

Header

The header of a data file consists of lines of ASCII characters. Its total length is determined by the parameter HeaderLen in the first line. The first line specifies general parameters, while the following define all states in the state vector, as well as all the current parameters. The number of bytes in the state vector is determined by the sum of the lengths (defined in bits) for all states, rounded up to the next byte (which equals the value of *StateVectorLength* in both the first line and, since *StateVectorLength* is also a system-wide parameter, in one of the lines in the `[Parameter Definition]` section). Thus, the data file can be read (although not fully interpreted) by reading the first line. The state definitions have the same format as described in 3.2.5 (although their values in the header are irrelevant, since they are defined for each sample in the data file). The parameter definitions have the format described in section 3.2.4. Since version 1.1, the first line begins with a *BCI2000V* field containing a version number in floating-point format, and ends with a *DataFormat* field describing the format of the binary data as *int16*, *int32*, or *float32*. A missing *BCI2000V* field indicates a file format version of 1.0, and a *DataFormat* of *int16*.

```
BCI2000V= 1.1 HeaderLen= l SourceCh= m StateVectorLength= k DataFormat= f CRLF
[ State Vector Definition ] CRLF
Name1 Length1 Value1 ByteLocation1 BitLocation1 CRLF
Name2 Length2 Value2 ByteLocation2 BitLocation2 CRLF
Name3 Length3 Value3 ByteLocation3 BitLocation3 CRLF
...
```

```
[ Parameter Definition ] CRLF
Section1 DataType1 Name1= Value1 DefaultValue1 LowRange1 HighRange1 // Comment CRLF
Section2 DataType2 Name2= Value2 DefaultValue2 LowRange2 HighRange2 // Comment CRLF
Section3 DataType3 Name3= Value3 DefaultValue3 LowRange3 HighRange3 // Comment CRLF
...
CRLF
```

The binary data directly follows this last CRLF.

Binary Data

For each sample, data values for all channels are stored, followed by *StateVectorLength* bytes for the state vector. Data samples are stored in little endian format depending on the *DataFormat* field:

DataFormat field	**data type**
int16	2-byte signed integer
int32	4-byte signed integer
float32	4-byte floating point (IEEE 754)

The number of samples in the file can be calculated as follows:

$$\text{samples} = \frac{(\text{file size in bytes}) - \text{HeaderLen}}{(\text{data value size in bytes}) \times \text{SourceCh} + \text{StateVectorLength}}$$

3.10.2 Parameter File

A parameter file consists of lines of ASCII in the same format as the parameter section in the data file header and as described in section 3.2.4.

3.10.3 BCI2000 File Extensions

The following table lists all extensions for BCI2000 data files:

description	extension
Data file	.dat
Parameter file	.prm
Application log file	.apl or .log
Statistics log file	.sta

3.11 Glossary

little endian
In little endian, the binary representation of a 16-bit integer is: low-byte (bits 0-7) first, followed by the high-byte (bits 8-15).

CR
stands for carriage return, or ASCII code 10.

LF
stands for line feed, or ASCII code 13.

CRLF
stands for carriage return and line feed, or (subsequent) ASCII codes 10 and 13.

parameter
A parameter is a static and global variable that describes some aspect of the system configuration. Parameters can only be created during the system startup phase, and changed during the information phase or when the system is in suspended state.

state
A state is a n-bit value that describes some aspect of the current system status. In communication, a state is represented as described in section 3.2.5.

state vector
The state vector holds all values for all states that are packed together in one array of bytes of length *StateVectorLength*.