PNNL-14342

Pacific Northwest National Laboratory

Operated by Battelle for the U.S. Department of Energy

# Final Report for the Energy Efficient and Affordable Small Commercial and Residential Buildings Research Program

# Project 3.3 - Smart Load Control and Grid Friendly Appliances

M. Kintner-Meyer R. Guttromson
D. Oedingen S. Lang

July 2003

Prepared for the U.S. Department of Energy
under Contract DE-AC06-76RL01830

PACIFIC NORTHWEST NATIONAL LABORATORY
*operated by*
BATTELLE
*for the*
UNITED STATES DEPARTMENT OF ENERGY
*under Contract DE-AC06-76RL01830*

**Printed in the United States of America**

**Available to DOE and DOE contractors from the
Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062;
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov**

**Available to the public from the National Technical Information Service,
U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm**

This document was printed on recycled paper.
(8/00)

# Final Report for the Energy Efficient and Affordable Small Commercial and Residential Buildings Research Program

# Project 3.3 – Smart Load Control and Grid Friendly Appliances

M. Kintner-Meyer
R. Guttromson
D. Oedingen
S. Lang

July 2003

Prepared for
the U.S. Department of Energy
under Contract DE-AC06-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99352

## Executive Summary

This report summarizes the results of research initiated in April 2000 under funding from the California Energy Commission and co-funding by the U.S. Department of Energy. The objective of this project was to develop, implement, and test new methods for detecting pre-cursors of impending problems in the California electric power grid. The approach pursued in the project utilized information that is measurable at the wall outlet anywhere in the California. The approach deliberately focused on methods that do not require communication from an outside source, but rather function fully autonomously by relying on a local frequency sensor that measures the frequency of the alternating current (AC) power supply at the wall outlet and some control intelligence that can ultimately be implemented at low cost in commonly used appliances for homes and businesses.

During the course of the project, two load controller prototypes were developed, built, and tested. The first load controller prototype responded to under-frequency events and rapid decay in the grid frequency. The controller was based on a personal computer (PC) platform with an Microsoft DOS operating system. The second load controller prototype was used for the statistical and spectral analysis of historic frequency data of known grid events. It was based on a PC with a Linux operating system that provided real-time controller capability as well as processing historic data read from a data file.

The first controller was designed to react to a grid event and then trigger a load to trip off line. We demonstrated the prototype in the laboratory. An under-frequency load shedding scheme implemented at end-use devices and appliances has great potential value associated with its ability to displace reserve generation capacity. This reserve capacity is required to be available during fast responses of unplanned generation and transmission outages. Instead of utilizing generation to correct a frequency error, control of loads could be used to achieve the same effect. Thus, the economic value of a frequency responsive load controller would be similar to that of spinning reserves.

In an attempt to extend the responsive nature of the first controller prototype, precursors of impending grid problems specific to California were explored on which to base the development of a more advanced autonomous load controller design. After consultation with transmission planning engineers of the CAISO, we analyzed the following two grid problems relevant for California: 1) dynamic stability problems during high power imports into Southern California from East of the Colorado River based on the SCIT (Southern California Import Transmission) nomogram, and 2) voltage stability problems in the San Francisco Bay Area during heavy AC/DC North-to-South power flows based on the T-116B nomogram. For these problems, we explored pre-contingency detection methods that were intended to trigger a load reduction in advance of an impending problem. We used detailed dynamic simulations of the Western power grid (WECC) and a simplified single-input-single-output model for selected locations in California to explore signatures in the AC frequency signal for high-stress and low-stress cases. We defined the low-stress case as a grid condition in which standard CAISO operating procedures were observed. The high-stress case was defined as a hypothetical case, in which the system was operated outside CAISO safe operation conditions.

For the dynamic stability (SCIT) cases, the simulation results revealed recognizable differences between the high- and low-stress cases in the frequency signal and its autospectrum for different locations throughout California. This finding gave rise to the formulation of a set of hypotheses that identified distinct differences in the dynamic behavior of the grid frequency as the power system transitions from a low-stress to a high-stress condition. The hypotheses postulated were:

1. Higher standard deviation in the frequency signal for the high stress case
2. Higher min-max range in the frequency signal for the high stress case
3. Higher amplitude in the autospectrum of the frequency signal for the high-stress case.

Contrary to the dynamic instability cases, the simulation of the voltage stability problems furnished results that revealed no differences in the dynamic behavior of the power system between the high- and the low-stress cases. These results led to the conclusion that the dynamic analysis approach appears not appropriate for voltage stability problems.

To test the hypotheses postulated, historic data representing two distinct grid events were analyzed. The first data set represented the WECC breakup of August 10, 1996, that caused wide-spread outages in the western region. The other data set (dated October 8, 2002) represented a transmission line trip followed by some remedial action and scattered load loss.

The results of the data analysis did not support our hypotheses. Finding some historic data that are representative of low- and high-stress conditions was difficult. The randomness and magnitude of constantly changing loads and adjustments by generators to meet the demand, coupled with the randomness of the unplanned outages, which cause changes in the topology of the network, makes it very difficult, if not impossible, to definitively declare a state of the power system as low stress. Even during periods at night, when the load tends to be lower than during the day, it is not obvious that the system attains a low or lower-stress state. Transmission outages, planned or unplanned, may pose a difficult burden on transmission engineers to keep the system in stable and safe condition. Because of the inherent inability to establish a state of low stress as a reference case, it became difficult during this analysis of historic data to detect the transition from a safe condition to that of an impending problem.

A necessary requirement for an effective detection technology is to recognize system conditions as the power system approaches dangerously close the edge of stable and safe operating conditions. Because of the complexity of the power system, the edge of safe operations is a moving target and depends on load conditions and network topology and thus may change from hour to hour.

As a result of this data analysis, it appears questionable whether the chosen approach will be successful in the long-run. The major obstacle for this approach is the necessity to establish a reference scenario that would represent safe grid operating conditions. To establish this, a large series of the conditions needs to be analyzed to become familiar with the spectrum of variability for each indicator to establish signatures or patterns for impending problems.

An alternative approach, if feasible, could potentially lead to a promising detection of dynamic instability of the power system. This alternative approach focuses on determining the transfer function that describes the dynamic behavior of the entire power system, from which the standard stability analysis methods can be applied. So far, no one has successfully established a power system transfer function of sufficient accuracy with which to perform a meaningful stability analysis.

With the insights gained from the simulation and data analysis, the following recommendations for additional research are made:

1. Under-frequency load control could provide an important grid reliability enhancement. Although reactive in its response, an under-frequency load control strategy with frequency responsive appliances and devices could provide reserves that are currently furnished by generators that are either already spinning or that can be ramped up in their output.

2. To enhance fundamental understanding of the stability characteristics of the power system, it is recommended that system identification techniques be used to approximate a real-time transfer function of the entire power system. If a real-time system transfer function of sufficient accuracy can be established, it would enable the use of standard stability analysis tools for determining distance to the stability edge.

3. For dealing with voltage stability problems, we recommend the use of under-voltage relays of induction motors, as found in compressor motors for air-conditioning systems and other appliances. The under-voltage protection prevents motor stalling caused by decreasing voltage as a result of a line fault or high system loading. The stalling of induction motors perpetuates the decreasing voltage to a point, where the voltage may drop sharply and quickly and propagate through the distribution systems as other electric motors reach the same conditions.

# Table of Contents

## List of Figures

## List of Tables

## Acronyms

| | |
|---|---|
| AC | alternating current |
| AEC | Architectural Energy Corporation |
| AGC | Automated generation control |
| AHU | air-handling unit |
| CAISO | California Independent System Operator |
| COI | California Oregon Intertie |
| dB | Decibel |
| DC | direct current |
| FFT | Fast Fourier Transform |
| FPGA | Field programmable gate array |
| FSU | Frequency sensor unit |
| GFA | Grid-friendly appliance |
| HVAC | heating, ventilation, and air-conditioning |
| Hz | Hertz (1/second) |
| ISO | Independent System Operator |
| kW | kilowatts |
| mHz | millihertz ($10^{-3}$ Hz) |
| ms | millisecond ($10^{-3}$ second) |
| MW | Megawatt |
| NERC | North American Electric Reliability Council |
| PCB | Printed circuit board |
| PDCI | Pacific Direct Current Intertie |
| RAS | remedial action schemes |
| RPM | Rounds per minute |
| SMUD | Sacramento Municipal Utility District |
| WECC | Western Electricity Coordinating Council (name was changed from WSCC in 2002) |
| WSCC | Western System Coordinating Council |

# 1 Introduction

This report summarizes the results of research initiated in April 2000 under funding from the California Energy Commission and co-funding by the U.S. Department of Energy. The objectives of this project were to develop, implement, and test new methods for detecting precursors of impending problems in the California electric power grid and to the extent possible develop autonomous grid-friendly appliance (GFA) controllers based on those methods. The approach pursued in this project utilized information that is measurable at the wall outlet anywhere in California. The approach deliberately focused on methods that do not require communication from an outside source, but rather function fully autonomously by relying on a local frequency sensor that measures the frequency of the AC power at the wall outlet and some control intelligence that can ultimately be implemented at low cost in commonly used appliances for homes and businesses.

During the course of the project, two load control prototypes were developed, built, and tested. The first load controller prototype responds to under-frequency[1] events and rapid decay in the grid frequency. This controller was designed to react to a grid event and then trigger a load to trip off line. We demonstrated the prototype in the laboratory. In an attempt to extend the responsive nature of the first controller prototype, precursors of impending grid problems specific for California on which to base more advanced autonomous load controllers were explored.

The proactive feature of a smart load controller was a particularly attractive research goal during 2000 and 2001, when California faced the significant shortfalls of generation capacity to meet demand. It became apparent during the early stages of this project that a control device that measures the grid's electrical properties at the wall outlet will never be able to sense the California Independent System Operator's (CAISO's) emergency stages (Stage I, II, and III) because they are determined by the ISO based on market data, not exclusively by the physics of the electric power system. As a consequence, the main research effort was re-focused on identifying grid stress as a precursor to power system outages in California. After consultation with transmission planning engineers of the CAISO, we focused on two grid problems of concern to the CAISO. For these problems, we explored pre-contingency detection methods that could trigger a load reduction in advance of an impending problem.

[1] An under-frequency event is an excursion in alternating current frequency below the nomimal value of 60 Hz.

## 2 Definition of Grid Stress

Throughout this report we refer to conditions in the electric power system that we call *grid stress.* Grid stress is a very broad term used to describe conditions where the regional transmission system is approaching a dynamically unstable condition. Many forms of grid stress may exist, such as voltage stress or stress which deteriorates small signal stability, with each type being heavily influenced by different initial conditions. In this project, we used the proximity of the grid to specific stability limits as expressed in CAISO nomograms as the definitions of grid stress.

The SCIT (Southern California Import Transmission) nomogram [CAISO 1998] identifies the safe operation of the Southern California grid as a function of total power imports into Southern California, Southern California power imports from East of the Colorado River only, and the Southern California System Inertia (with units of Megawatt seconds or MWS). Operation within the boundaries of the nomogram is required for safe operation of the power system. If the system were operated outside of the bounds of the nomogram, then a worst-case disturbance[2] would cause power oscillations ultimately resulting in a power system breakup and a large-area loss of power.

There are other mechanisms that lead to grid stress as well. Some are known and accounted for within the design process of power systems, and others are not. Although the focus of this report is on two mechanisms of grid stress, it is implied that the concept of stress detection can be applied more broadly.

[2] Often referred to as the N-1 event.

# 3 Development of a 1st Generation Load Controller

## 3.1 Objective of First Generation Load Controller

The objective of the simplified load controller developed in this project was to provide a fast-responding control device that sheds load in response to a grid event. The controller device senses the local grid frequency at the wall outlet and de-energizes the load, based on two criteria. The criteria are: (1) frequency below a specified threshold (an under-frequency event) and (2) rate of frequency decay is greater than a user-specified threshold. The load is re-energized after the load shedding criteria are no longer met, utilizing a random waiting period between 0 and 1 minute to ensure that individual appliances re-energize at slightly different times providing a smooth transition for the grid.

### 3.1.1 What Electric Grid Events Can Be Detected in the Grid Frequency?

The grid frequency in any power system is automatically controlled at the generator to maintain the nominal 60 Hz frequency. As a loads are turned on and off, generators respond to the resulting load changes. Mismatches between generation and load cause the generator to slow down or speed up. Since generators are synchronized to the grid, the change in speed of a generator causes changes in system frequency. In order to maintain a constant frequency of 60 Hz (or any other frequency set point determined by the power control centers) automatic frequency controls (speed governors) at the generator are required. Because sharp changes in grid frequency indicate imbalances between generation and load, the change in frequency is considered to be the sole input signal needed to detect the following major grid events:

- tripping generation off line
- major load switching (on or off)
- unscheduled tripping of transmission lines (interruption of power flow to loads)
- exhausted ancillary reserves

#### 3.1.1.1 A Tripping Generator

A generator trip of sufficiently large capacity (i.e., several hundred MWs) causes an immediate imbalance between the generation and the load. Instantaneously, the power for the new load is supplied by all other generation attached to the grid. This is accomplished automatically by the laws of conservation of energy. The new power delivered comes from the rotational kinetic energy of the generators attached to the electric grid, and results in a decrease of speed (and frequency) until speed governors for these units are able to increase their mechanical power input and match the new load.

Depending on the amount of generation lost, the amount of inertia on the grid, the amount of reserve generation available, and frequency response characteristics of all sped governors, the frequency drop may be small (e.g., a few mHz) and brief (a few seconds) or severe (~100 mHz) and prolonged (over a minute in duration). Figure 3-1 shows a frequency excursion caused by a generator trip and recovery back to the nominal 60 Hz frequency after 10 seconds.

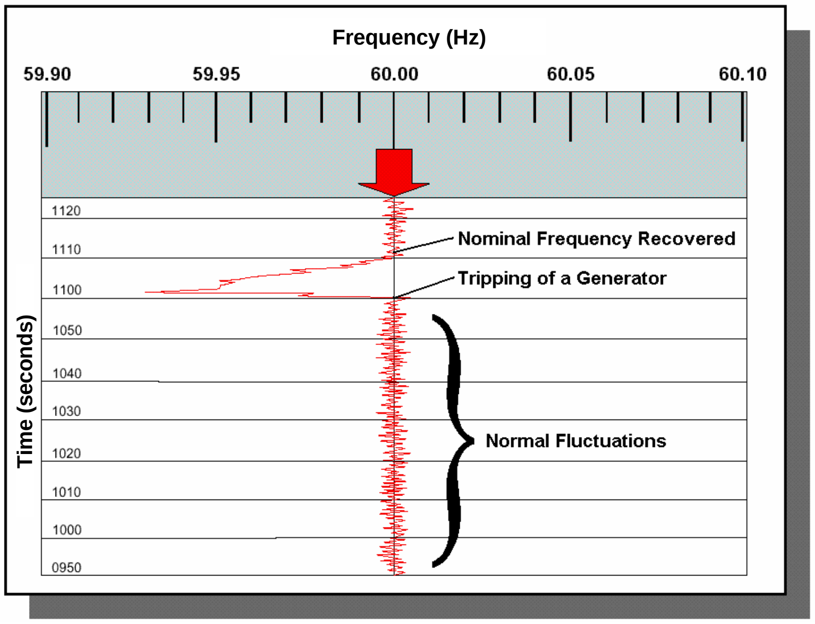

Figure 3-1: System Frequency Response to the Loss of a Generator.

### 3.1.1.2 Tripping a Transmission Line

A transmission line trip is generally caused when a line is short circuited because of a lightning strike, shorting out by trees, or by other relay protective actions. The line break typically causes a load loss, resulting in a brief over-generation, causing the grid frequency to increase. The opposite may also occur, in which case, more generation is removed than load, resulting in an under-generation condition, causing the grid frequency to decay.

### 3.1.1.3 Exhausted Ancillary Reserves

Power systems planning engineers design the power system to withstand, at minimum, a single-point failure (called an N-1 contingency). Although the system is designed to survive any single-point failure, the dispatch of generation reserves protects the system from more serious system casualties and provides additional margin for unanticipated load or lost generation. During situations of strained resources, however, when, for instance, multiple generation outages occur, the frequency excursion can be particularly severe (i.e., causing a large frequency drop) and recovery from it, back to normal conditions, may be prolonged over several minutes.

### 3.1.2 Design Considerations for Local Frequency Measurements

The typical dead band of a speed governor's frequency control of about 0.036 Hz was used as a guide for determining the needed frequency resolution of our sensor [Kirby 2003, Hirst et al. 2003]. According to the North American Electric Reliability Council (NERC) policy, generators greater than 10 MW in rating should operate utilizing frequency responsive governors. These governors should be capable of providing immediate and sustained response to abnormal frequency excursions, providing a 5% droop characteristic, and fully responsive to frequency deviations exceeding ± 0.036 Hz (± 36 mHz). [NERC 2002].

For this analysis, a sampling rate was used that provided a resolution of 0.036 Hz as an upper bound to meet the accuracy requirements. It is desirable to have a higher sampling rate to differentiate random fluctuations within the generator's dead band from those that are affected by generator control action. Practical considerations determined by the cost to build and calibrate the sensor limited the resolution. Furthermore, there is a trade-off between resolution and measurement range given memory constraints of microprocessors. This limited the resolution of the sensor to about 1 to 2 mHz for a measurement range of ± 100 mHz about a nominal 60.0 Hz AC grid frequency.

## 3.2 Design and Implementation of the 1st Generation Load Controller

Commonly deployed standard frequency counters for laboratory use, like the HP 53131A device from Hewlett Packard [HP 2003] or the HAMEG HM8021 [HAMEG 2003], are not adequate for a prototype implementation. They use long periods, 1 second or more, to achieve the required resolution at low frequencies (< 100 Hz). The results presented by these counters are average values and, therefore, inappropriate for this analysis, since we need to quickly detect frequency changes (i.e., within two or three 60 Hz-cycles).

Most industrial products that specialize in higher resolution frequency measurements have a typical accuracy of about 0.1% (0.1% of 60 Hz is 60 mHz). This is insufficient for measuring the frequency excursions typically observed during grid events. Most of these frequency sensors cannot measure the frequency within one or two cycles and are generally designed to measure high-frequency signals (e.g., audio frequencies, rounds per minute (RPM) of engines, and radio frequency signals).

Utility-grade frequency meters used for under-frequency load shedding at substations are very expensive (>$10,000) and typically offer extra features that are not applicable. The costs would be prohibitive for use on appliances.

### 3.2.1 Block Diagram of Simplified Load Controller

Figure 3-2 shows the most important functional units of a frequency sensor unit (FSU) attached to a load.

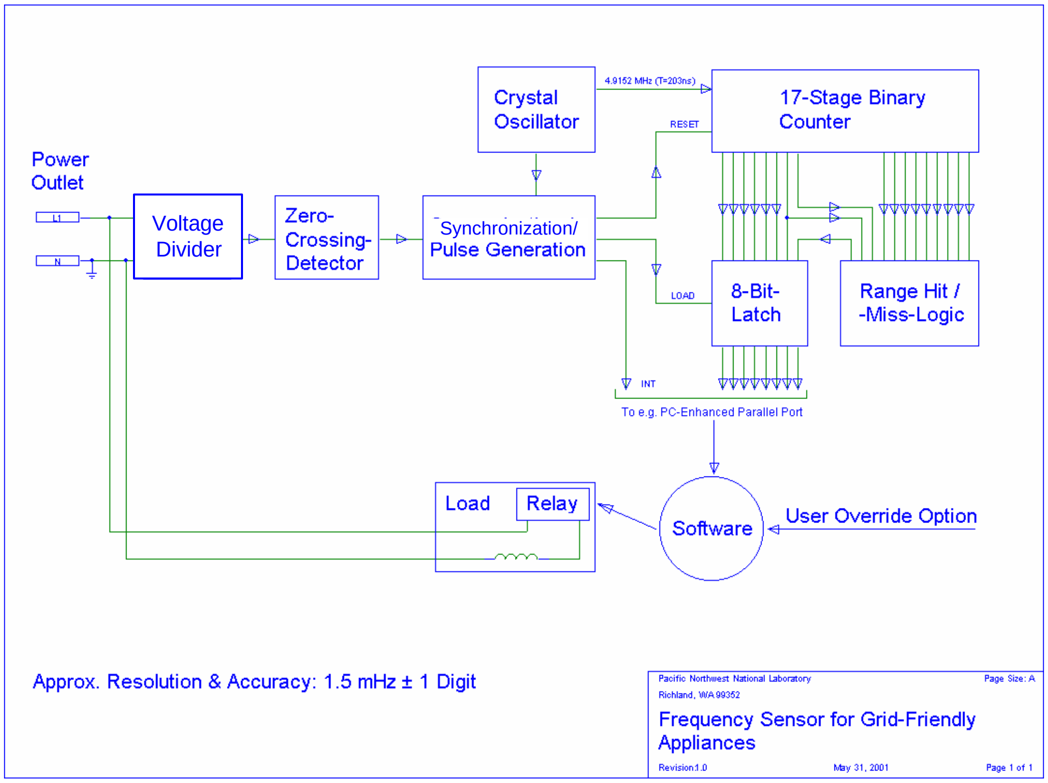

Figure 3-2: Functional Units of a Frequency Sensor Unit

### 3.2.2 Frequency Sensor and Controller Description

Figure 3-3 illustrates the major functional components of the 1st generation load controller, including the frequency sensor, PC as controller, the relay, and the load. A complete and detailed list of the sensor components is provided in Appendix A: Description of Frequency Sensor.

The features of the frequency sensor are:

- Hardware data acquisition every 16.67 ms (60 samples/second)
- Hardware averaging of last six values.
- Sensor range: 60 Hz ± 100 mHz
- Resolution: 1.5 mHz
- Accuracy: ± 1.5 mHz

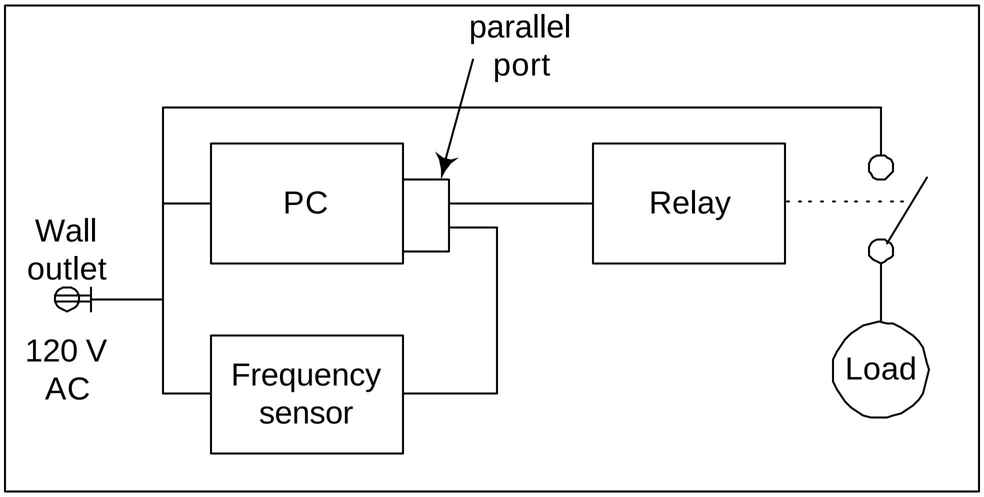

Figure 3-3: Block Diagram of 1st Generation Load Controller

- Interface: 8-bit parallel for PC-enhanced parallel ports or any micro-controller
  - 7 data lines
  - 1-bit hardware used to communicating frequency outside the specification range (60 Hz ± 100 mHz)
- Required inputs: Grid voltage (line, neutral); ±9 V DC (20 mA each)
- Interrupt signal generation available
- Costs: components about $10 (at sample quantities) plus cost for printed circuit board of $10 in large volume production.
- Physical dimensions of PCB (length x width x height):
  123.5 mm x 56.0 mm x 18.0 mm (4.86 in. x 2.20 in. x 0.71 in.) including connectors.

### 3.2.3 Prototype of 1$^{st}$ Generation Load Controller

The prototype is a standalone load controller combined with a frequency sensor. A single- board PC made by Advantech (Advantech BiscuitPC processor board) [Advantech 2003] monitors frequency and performs the load control action utilizing a solid-state relay that supports loads up to 1.5 kW. There are two power receptacles at the back of the prototype (see Figure 3-4).

The PC runs on an MS-DOS operating system and boots up automatically when power is turned on. A small, 20 x 8 character LC-display[3] shows system status. Three programming buttons allow the user limited resetting of load shedding threshold values.

[3] Eight-line serial LC-display (SEETRON G12864 V2.0, see http://www.seetron.com/slcds.htm for details).

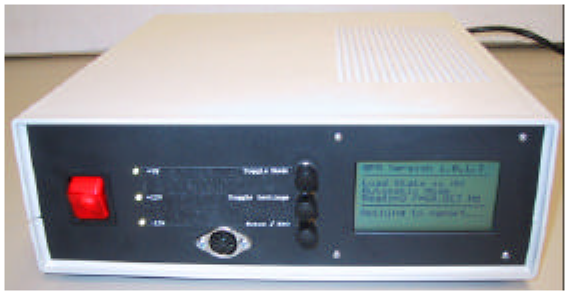
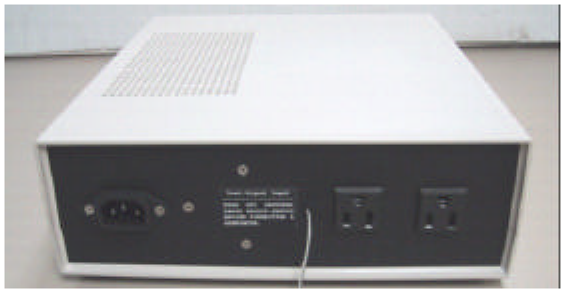

Figure 3-4: Front- and back-panel of the 1st Generation Load Controller

The three push-buttons have the following functions:

- **Button 1:** Toggle Mode; toggles between the automatic and override modes. In automatic mode, the load controller monitors the grid frequency and opens the relay if the user-definable frequency threshold criteria are met. In the override mode, the user closes the relay to override a load curtailment.
- **Button 2:** Toggle Settings; push to display the load shedding criteria (i.e., over- and under-frequency criteria and rate of frequency decay criterion)
- **Button 3:** Enter/ESC; confirms messages and exits the program.

For programming, a PC-keyboard connector is provided.

A photograph showing the major components inside the prototype is shown in Figure 3-5.

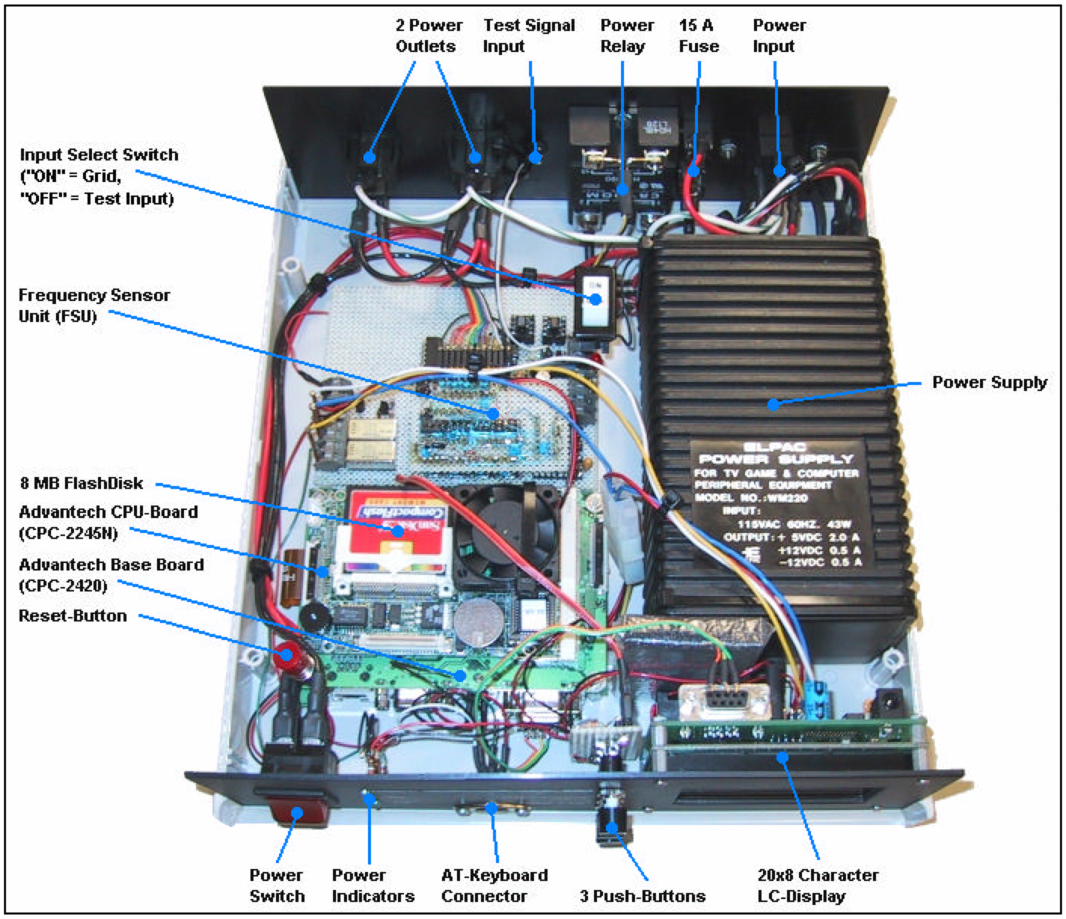
2 Power Outlets
Test Signal Input
Power Relay
15 A Fuse
Power Input
Input Select Switch ("ON" = Grid, "OFF" = Test Input)
Frequency Sensor Unit (FSU)
Power Supply
8 MB FlashDisk
Advantech CPU-Board (CPC-2245N)
Advantech Base Board (CPC-2420)
Reset-Button
Power Switch
Power Indicators
AT-Keyboard Connector
3 Push-Buttons
20x8 Character LC-Display

Figure 3-5: Inside View of 1st Generation Load Controller Prototype

### 3.2.4 Software Description

The software to evaluate the grid frequency signal has been written completely in Borland Turbo C v. 2.01 for MS-DOS and can be executed on an ordinary PC or compatible running one of the following operating systems [Groll et al. 1998]:

- MS-DOS 5.0 or higher
- Microsoft Windows 3.x
- Microsoft Windows 95/98/ME.

The software (current version is 1.0.1.3) provides two basic modes of operation:

- GFA Load Control Logic Mode
- Data Logger Modes for Data Acquisition Purposes.

Both versions can currently take advantage of the LPT1 parallel port of any PC or the Advantech BiscuitPC processor board, as shown in Figure 3-5. The software code is given in Appendix B of this report.

## 3.3 Load Controller Capabilities

Control software was developed to retrieve frequency data from the sensor at a rate of 10 samples per second. The frequency data are compared to the following user-definable load curtailment criteria:

1. Under-frequency criterion: grid frequency < 59.95 Hz, the default set point.
2. Rapid frequency-decay criterion: rate of frequency change < -0.125 Hz/sec, the default set point

The criteria above were independently tested. If one of the above criteria is met, the load is de-energized. If triggered by an under-frequency event, two conditions must be met before the load is turned on again. First, the frequency must recover by a user-definable value above the under-frequency set point (default is 10 mHz) <u>and</u> after a time lag of random duration between 0 and 30 seconds, the load is reconnected. The random time lag prevents all loads from turning back on all at the very same time after a frequency event, which could cause a rebound effect, potentially tripping other overload relays on transmission or generation equipment.

If rapid frequency decay triggers a load curtailment, the load is reconnected after a random time lag following the time when the frequency-decay criterion is no longer met.

Detailed information on the controls software can be found in Appendix B.

## 3.4 Value of 1st Generation Load Controller

The 1st generation load controller is responsive to imbalances between generation and load that manifest themselves in reductions in grid frequency as the turbines and generators are slowed down. This condition can occur if very large loads (e.g., arc furnaces) are rapidly turned on or if generation or transmission capacity is tripped off line. In most cases with sufficient generation reserves, the frequency will recover to its normal set point as generation is redistributed across a large interconnected power system, meeting the load. An under-frequency load shedding scheme implemented at end-use devices and appliances has great potential value associated with its ability to displace reserve generation capacity. This reserve capacity is required to be available during fast responses of unplanned generation and transmission outages [Kirby 2003]. Instead of utilizing generation to correct a frequency error, control of loads could be used to achieve the same effect. Thus, the economic value of a frequency responsive load controller would be similar to that of spinning reserves.

# 4 Development of Data Analysis Platform

## 4.1 Introduction

Two types of data were analyzed in this study, data from simulation of the electric power grid and empirical data collected from the actual grid during operation (including during well-known grid events). In support of analyses of field data, hardware and software tools were developed that enabled us to develop and test detection methods and new load shedding algorithms. We specified that the tools be versatile to be used for real-time analyses as well as for analyses of historic data. Two data input streams were implemented to process: (1) real-time data measured at the wall outlet and (2) historic data of known grid events stored in data files.

## 4.2 Analysis Tool Platform

The analysis tool platform consists of a personal computer with a special purpose ISA card. The ISA card contains a frequency sensor system and a solid state relay for load shedding. The analysis tools are implemented in ANSI C programs executed on a Linux operating system [ANSI 1988]. Data for the analysis can be provided either by the frequency sensor in 100 ms intervals in real time or by reading from a data file. The functional blocks of both hardware and software are shown in Figure 4-1.

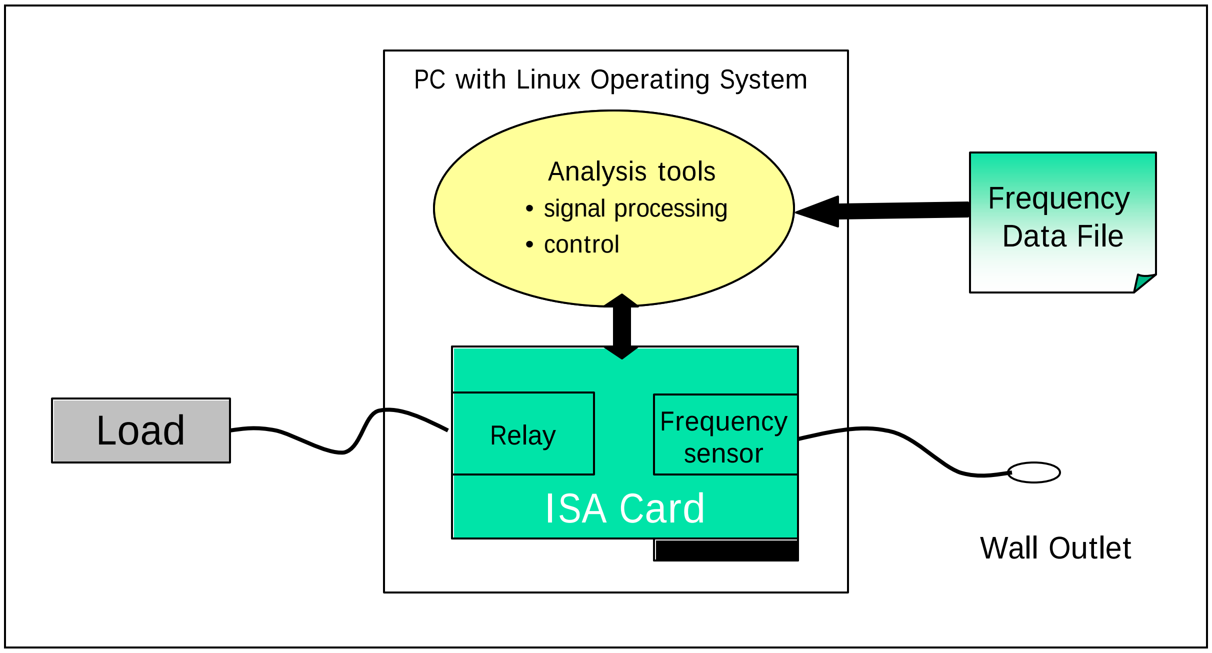

Figure 4-1: Schematic of Tool for Data Analysis

### 4.2.1 Hardware

The hardware used for this project is a personal computer with a multi-tasking Linux operating system that provides data acquisition of grid frequency signals and real-time signal processing capabilities.

The computer is a Dell with a 400 MHz Pentium processor and 64 MB of internal memory, running a *Redhat Linux*[4] operating system. The computer has a special-purpose ISA card that accommodates a frequency sensor for measurements of the AC power-supply frequency and a solid state relay that opens and closes a contact to a small load. The solid state relay can be accessed by software for load shedding purposes. The specifications of the frequency sensor are identical to those discussed in Section 3.2.2. The logic and hardware implementation was optimized to fit into a field programmable gate array (FPGA) architecture that reduced the size to a 2 by 3 inch section on the board. We used the MAX 7000A model designed by the Altera Corporation for the FPGA [Altera 2003].

### 4.2.2 Software

All necessary Linux drivers to perform real-time data acquisition were developed (see Appendix B). Linux drivers are executable software modules that are loaded into the Linux kernel during PC start-up and, thus, become part of the operating system functions. The signal processing software was developed as an application program and as such needed to be called by the user after the operating system was loaded. Both Linux drivers and signal processing routines were written in ANSI C programming language, utilizing the C compiler resident in the Linux operating system [ANSI 1988]. Detailed information on the controls software of the data analysis platform can be found in Appendix B.

## 4.3 Reading Real-Time Data

In the real-time mode, data are read from the frequency sensor. The Linux driver reads 256 frequency data points each time it requests data from the sensor. With a sampling frequency of 10 samples per second (read at 100 ms time intervals), 256 samples comprise a time period of 25.6 seconds. A sliding time window of 25.6 seconds length was implemented that advanced in time every 100 ms, receiving 10 new data points and discarding the 10 oldest data points.

The real-time data acquisition and the synchronization with analysis programs were tested to assure that the analysis is completed before the next updated data set is processed. Execution of the analysis routines was found sufficiently fast to finish before

[4] Version 6.1 released on October 07, 2000.

the next batch of data is read. An alarm messaging was implemented to alert the user to potential time conflicts in case the analysis took too much time and prematurely aborted because the next data retrieval started.

## 4.4 Algorithms

Several software routines were developed for analysis of the grid frequency signal. The selection of routines was guided by the findings of the simulation results in Section 4. We hypothesized that impending problems of high-stress conditions are potentially detectable by: (a) greater absolute maxima in the gain of the spectrum, (b) overall greater absolute gain values of the spectrum in the relevant frequency range between 0 and 2 Hz, and (c) sharper maxima, compared to low-stress conditions.

To capture these characteristics in the spectrum, we established analytical tools consisting of software routines that would perform the following rudimentary functions:

- Generation of the spectrum
- Determining the maxima in the spectrum
- Determining area under the spectrum by frequency bands
- Determining sharpness or pointedness of the spectrum at the locations of the maxima.

Sections 5.4.1 through 5.4.5 provide brief overviews of the analytical tools.

### 4.4.1 Development of Spectrum of a Signal

The discrete Fast Fourier Transform (FFT) algorithm as described in [Press et al. 1993] was used to compute the spectrum of the grid frequency signal. Of interest was only the magnitude of the spectrum to indicate oscillatory content of the signal, not the phase angle of the signal. The magnitude of the spectrum was computed in terms of commonly used decibel (db) notation.

### 4.4.2 Spectral Bands

The interesting range of the spectrum is between 0 Hz and 2 Hz. Frequencies above 2 Hz are likely to contain less information on the system's oscillatory behavior and are more likely to contain higher random noise contributions. Because the 0 to 2 Hz bandwidth is relatively large for detecting changes as a function of time, we segmented this frequency range to isolate frequency regions of interest. Table 4-1 shows the five bands into which it was segmented. The frequency range within each band was represented by 10 discrete data points with an equal distance of 0.04 Hz between them.

Table 4-1: Definition of Bands in the Spectrum

| Band | Frequency range |
|---|---|
| Band 1 | 0.0 Hz – 0.4 Hz |
| Band 2 | 0.4 Hz – 0.8 Hz |
| Band 3 | 0.8 Hz – 1.2 Hz |
| Band 4 | 1.2 Hz – 1.6 Hz |
| Band 5 | 1.6 Hz – 2.0 Hz |

### 4.4.3 Integral

The first characteristic of the frequency signal is the integral of the spectrum. It can be interpreted as a general indicator of the oscillatory content in a frequency band. A routine that determines the area under the spectrum in each frequency band was established.

### 4.4.4 Maxima

A function was implemented that locates local maxima in any function and determines the absolute value of each maximum. The local maximum is found if the first derivative is zero and the second derivative is negative.

### 4.4.5 Sharpness Detection

#### 4.4.5.1 Definition of Sharpness

The measure of sharpness of a curve at its maximum or multiple maxima was defined in terms of a change in the slope of a curve at its maximum. The magnitude of the second derivative at the maximum of a curve is directly related to the curve's sharpness and was used as a sharpness index. Another, perhaps more intuitive definition, was used that defines an angle spanned by the peak. Both definitions were used and implemented.

Figure 4-2 schematically shows three variants of how the peak in a discrete signal can occur. We investigated the importance of considering a five-point peak ensemble (see Figure 4-2b) versus a three-point ensemble (Figure 4-2a and c). According to our analysis of the spectra from historical grid frequency data, the scenario in Figure 4-2b rarely occurs. Most common peaks are similar to that shown in Figure 4-2-c. As a result, we established a three-point peak-finding routine for determining the sharpness of peaks.

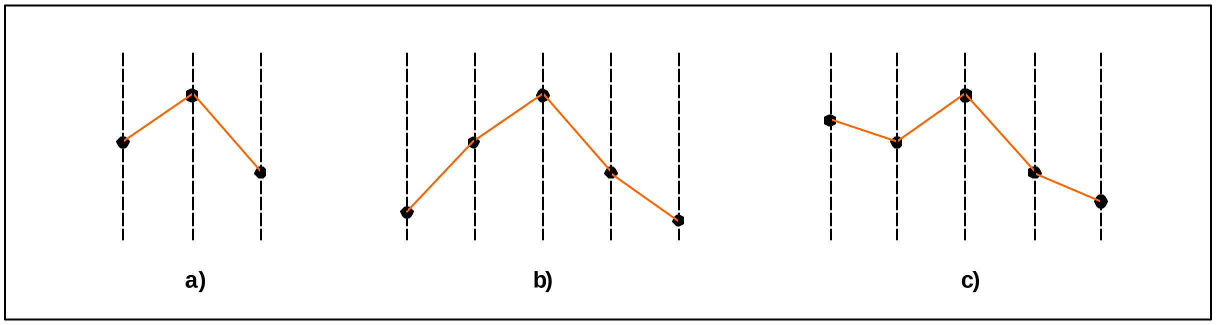

Figure 4-2: Definition of Sharpness at the Maximum of a Discrete Signal

#### 4.4.5.2 Definition of Sharpness Using Second Derivative

Figure 4-3 displays the discrete signal, and its first and second derivatives. We chose a forward-differencing strategy that assigns the difference of two nodes to the first node. Adopting this approach, the second derivative is then assigned to the first node of the three-point peak ensemble as shown in
Figure 4-3.

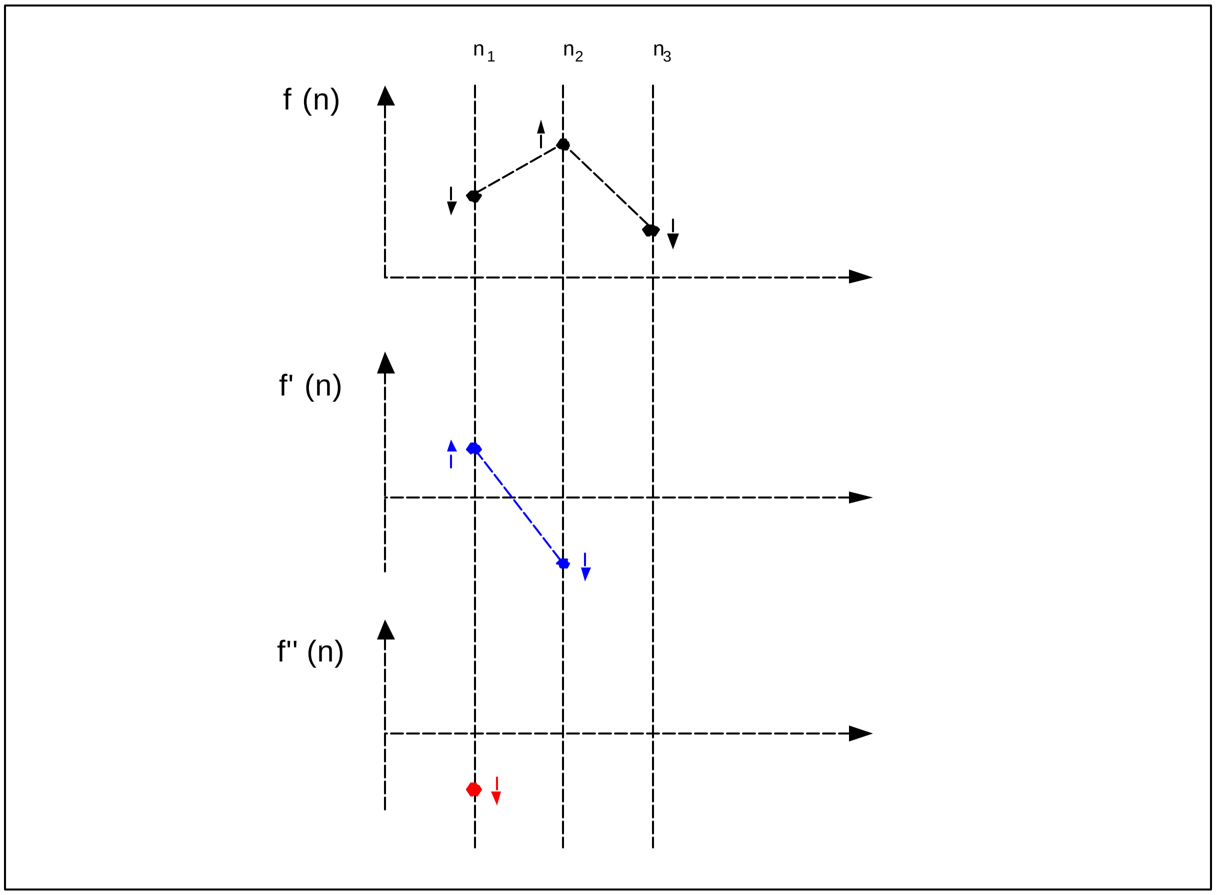

Figure 4-3: Second Derivative Used for Determining the Sharpness at the Maximum of a Discrete Signal

#### 4.4.5.3 Definition of Sharpness By Angle

Another method of measuring sharpness is by calculating the angle at the peak, as shown in Figure 4-4.

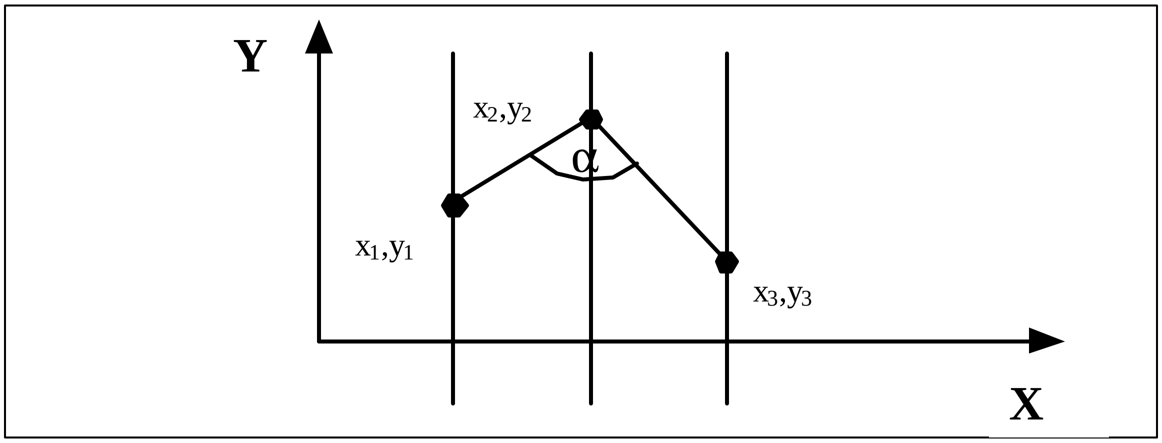

Figure 4-4: Definition of Sharpness at the Maximum of a Discrete Signal by Determining the Angle

The angle is determined by calculating the angle spanned by the slopes to the right and left from the peak using the following triangular relationship:

$$a = \arctan\left(\frac{x_2 - x_1}{y_2 - y_1}\right) + \arctan\left(\frac{x_3 - x_2}{y_3 - y_2}\right).$$

# 5 Analysis of Grid Stress

This section presents the analysis of grid behavior and development of grid-stress detection methods based on simulations of the electric power grid.

## 5.1 Motivation for Enhanced Load Controller for Detection of Grid Stress

The frequency-based load controller responds to declining grid frequency. This condition is generally encountered when generation capacity or major bulk power transmission lines are disrupted. Other grid-stress conditions, however, exist that potentially can lead to overload conditions on the transmission system. Detection of precursors to these conditions so that grid-stress events could be anticipated or even prevented would be quite valuable, but detection of impending events is very difficult and little is known about this today. These are the conditions that lead to a generation or transmission outage rather than those that are caused by an outage. Examples are oscillatory behavior of bulk power flows or voltage collapse.

Because of the recognition that oscillatory power flow and voltage collapse conditions are of high importance to the California power system, these conditions were investigated further to learn more about how they manifest themselves at wall outlets throughout California. The goal was to design signature detection algorithms for identifying these conditions. To this end, a simulation study of the Western Electricity Coordinating Council (WECC) of which California is a part (see Figure 5-1), was undertaken to analyze the power system under various conditions. In particular, dynamic stability conditions, which limit the imports from the Southwest into Southern California, were analyzed. These conditions are well known to the California Independent Systems Operator (CAISO) and regional transmission system operators. Transmission planning engineers developed operating nomograms that specify safe operating ranges for the California power grid under various load conditions. These nomograms are utilized to define two very different operating conditions that represent: 1) a very low stress operating point and 2) a very high stress condition. A sequence of simulations was used for the low and high stress conditions and data at various nodes in the California power system were analyzed. The results were then compared for significant differences, which were used to find a power system signature for identifying an impending operational issue.

## 5.2 Dynamic Stability Issues

The following section describes the low and high stress conditions that were analyzed to determine if the onset of a violation of the Southern California Import Transmission or SCIT nomogram [CAISO 1998] could be detected using a system signature.

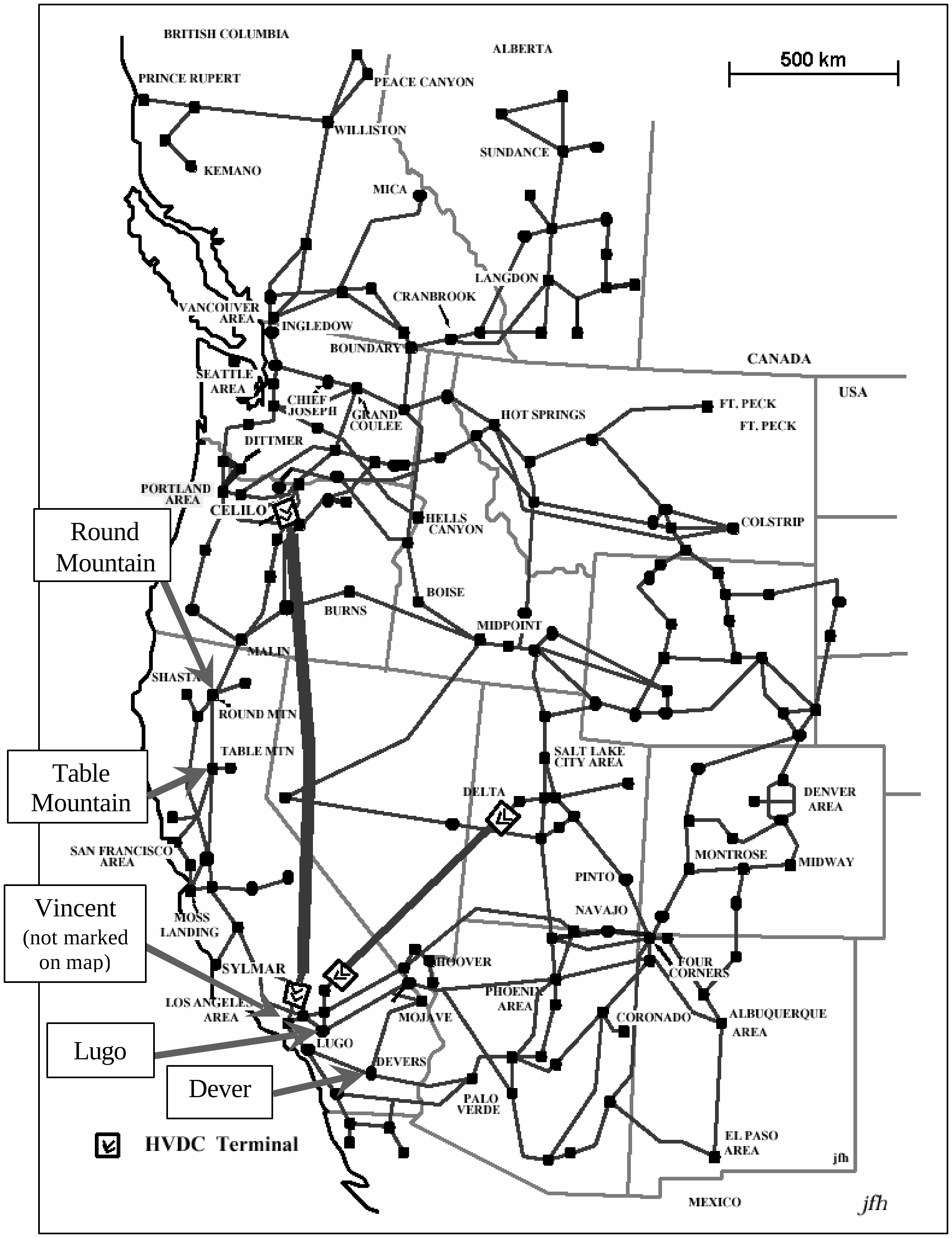

Figure 5-1: Major Transmission Lines of the Western Electricity Coordinating Council (WECC)

### 5.2.1 Establishment of Grid Stress Conditions - Definition

Grid stress is a very broad term used to describe a condition where the regional transmission system is approaching a dynamically unstable condition. Many forms of grid stress may exist, and each may be heavily influenced by different initial conditions.

In this project, the nearness to one specific stability limit was used as an indicator of grid stress.

One of the specific stability limits is defined by the SCIT (Southern California Import Transmission) nomogram [CAISO, 1998]. This nomogram identifies the safe operation of the Southern California grid as a function of total power imports into Southern California, Southern California power imports from East of the Colorado River only, and the Southern California System Inertia (in units of Megawatt seconds or MWS). Operation within the boundaries of the nomogram is required for safe operation of the power system. If the system is operated outside of the bounds of the nomogram, then a worst-case (N-1) disturbance would cause power oscillations ultimately resulting in power-system breakup and a large-area loss of power.

The two scenarios shown in Figure 5-2 were simulated. The low stress case is inside and the high stress case outside the stable envelope for operation. A system breakup would occur for the case outside the envelope (point H) if and only if a worst case N-1 disturbance were to occur, whereas the case operating inside the SCIT curves would remain stable under such a worst case system disturbance. For analysis of these cases, only a very small disturbance was initiated, not enough to cause instability, but enough to capture the system ringdown response for the two cases shown. The results could then be used to quantify the damping of the model for both cases.

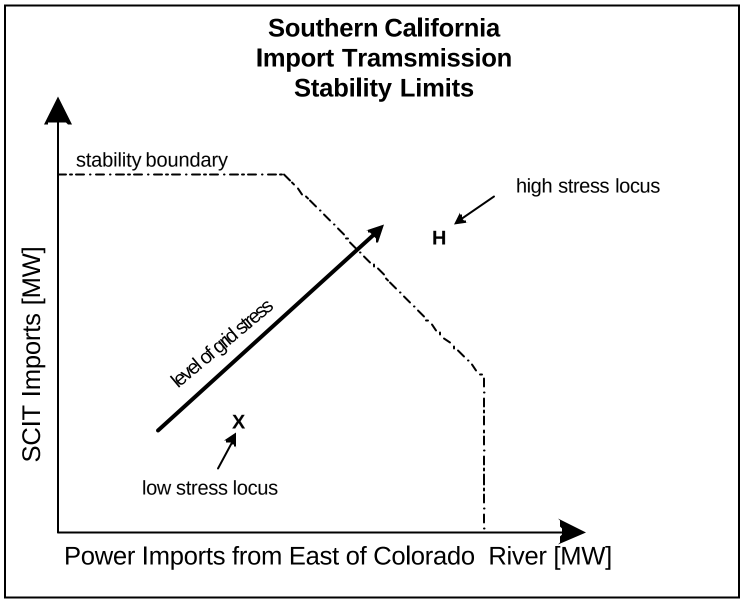

Figure 5-2: Southern California Import Transmission Nomogram. Locus X inside the stability boundary is considered low stress. Locus H outside the boundary is considered the high stress condition.

### 5.2.2 Analysis of Grid Stress in the Power System

We explored how the different degrees of grid stress could manifest themselves in the transmission system throughout the entire Western power system. To do this a set of power system stability simulations of the Western power system for high and low grid stress cases was performed. The results from these simulations were analyzed with respect to their dynamic behaviors and compared.

#### 5.2.2.1 Simulation

The dynamic stability simulation was performed for the WECC transmission grid. The simulation used a full model of the western North American power system developed by WECC member utilities [GE 2001]. Stability studies are typically performed by inducing an event such as a power demand spike in the system, which triggers a significant system response. The response is then analyzed. For the power system of the WECC, this has traditionally been done by inducing an instantaneous power demand of 1400 MW for 0.5 seconds followed by an instantaneous drop of the same magnitude at the Chief Joseph power station in the State of Washington (generally referred to as a Chief Joseph Brake Insertion). The system impulse response represents the system itself, and is then analyzed for both the low and high stress conditions.

The following steps detail our simulation approach:

1. Construct two dynamic models of the WECC system, representing high and low grid stress cases. The simulation was performed using General Electric's Positive Sequence Load Flow (pslf) simulation environment [GE 2001].

2. Quantify the level of grid stress in each model by modeling the Chief Joseph Brake and analyzing the damping of the oscillatory modes of the power system. The modes and damping of the power system for the two cases were determined using a Prony analysis technique.

3. Convert the frequency versus time response into a single input/single output linear model, $G_{pf}(s)$ transfer function, with power as input and frequency as output (see Figure 5-3). The transfer function is valid only for power inputs and frequency responses at their original locations.

4. Create a power (Gaussian white noise with constant power density/frequency) vs. time signal to represent and simulate system noise attributable to random phenomena, such as generators and loads being turned on and off-line.

5. Input the power (noise) signal into the single input/single output linear model and obtain the resulting frequency versus time signal from the model output. This signal is now assumed to represent the local "wall outlet" frequency we might directly measure at a residence's 120 V wall outlet. While the single input/single output model with the transfer function $G_{pf}(s)$ strictly simulates the system frequency as a

function of power at a specific substation in the transmission grid, we assume that the frequency changes across transformers downstream into the distribution systems to the end-use devices are negligible. Under this assumption the frequency at the point of analysis is identical to the frequency seen at the wall outlet.

6. Characterize the frequency signal using:
    a. Min–to-Max amplitude band
    b. Standard deviation of amplitude
    c. Minimum or maximum rates of change in frequency amplitude
    d. Peaks in the Fourier Transform of the frequency response.

7. Compare the results from characterization of the frequency signals to the known system stress and develop correlations regarding system stress and various frequency characteristics.

The linear single input/single output is illustrated in Figure 5-3.

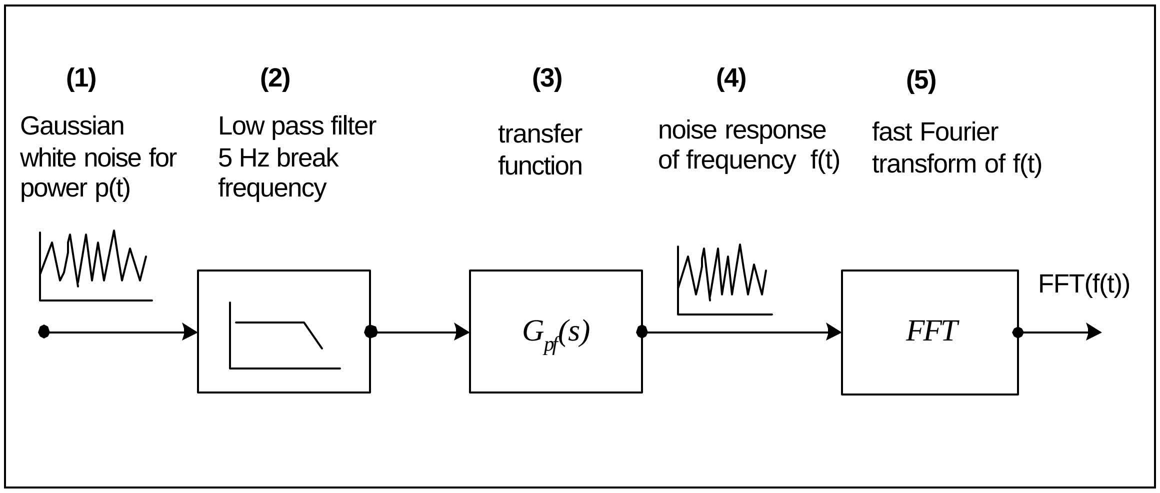
(1)
Gaussian white noise for power p(t)
(2)
Low pass filter 5 Hz break frequency
(3)
transfer function
(4)
noise response of frequency f(t)
(5)
fast Fourier transform of f(t)
$G_{pf}(s)$
FFT
FFT(f(t))

Figure 5-3: Linear single input/single output model with (1) noise power p(t) input, (2) low pass filter with 5 Hz break frequency, (3) transfer function, (4) noise response of frequency f(t), and (5) Fast Fourier Transform or spectrum of frequency f(t).

### 5.2.2.2 Simulation Results

The simulation results are shown in Table 5-1 and Figures Figure 5-4 through Figure 5-9. They indicate clear differences between the high and low stress cases in the system frequency response to the Chief Joseph Brake for different locations throughout California. The differences were all consistent. They showed the following characteristics:

- Higher standard deviation in the frequency signal for the high stress case (see Table 5-1).
- Higher min-max range in the frequency signal for the high stress case (see Table 5-1).
- Higher and for some peaks sharper maxima in the system transfer function $G_{pf}(s)$ for the high-stress case (see Figure 5-4, Figure 5-6, and Figure 5-8).
- Higher magnitude in the autospectrum of the frequency signal for the high-stress case (see Figure 5-5, Figure 5-7, and Figure 5-9).

Table 5-1: Comparison of standard deviation and maximum-to-minimum (max-min) range of frequency in high- and low-stress cases for three California locations.

| **Locations** | **High Stress** | | **Low Stress** | |
|---|---|---|---|---|
| | **Standard Deviation** | **Max-Min** | **Standard Deviation** | **Max-Min** |
| Lugo, CA | $10.1 \times 10^5$ | $7.7 \times 10^4$ | $7.3 \times 10^5$ | $5.4 \times 10^4$ |
| Vincent, CA | $9.8 \times 10^5$ | $7.8 \times 10^4$ | $6.9 \times 10^5$ | $5.3 \times 10^4$ |
| Devers, CA | $10.4 \times 10^5$ | $7.8 \times 10^4$ | $7.6 \times 10^5$ | $5.7 \times 10^4$ |

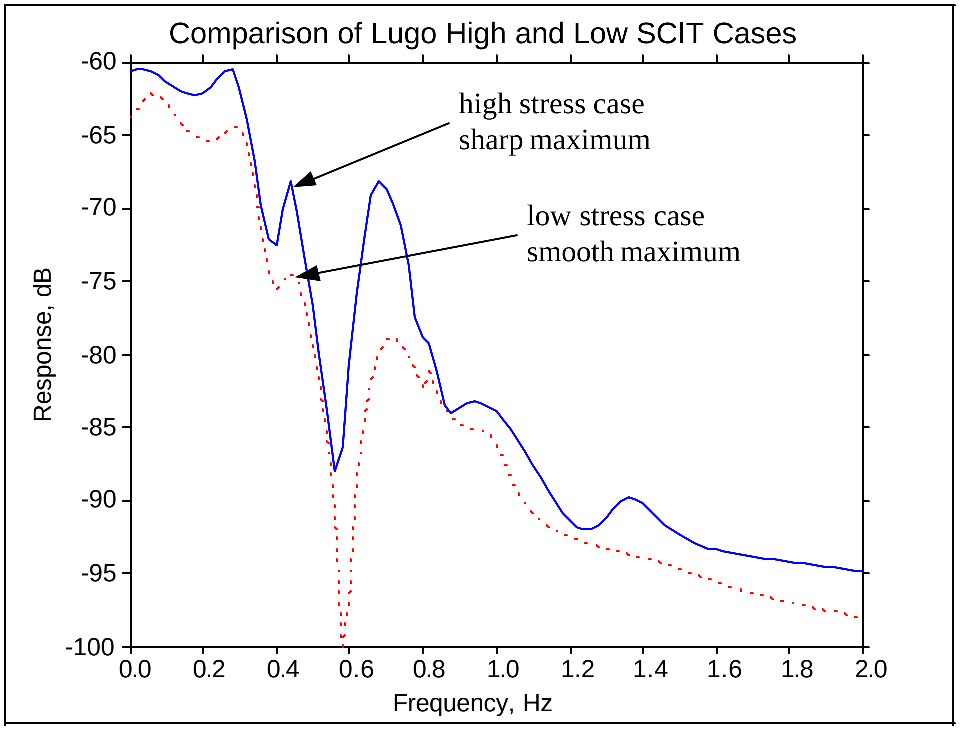

Figure 5-4: Response at Lugo, California, to Chief Joseph Brake event.

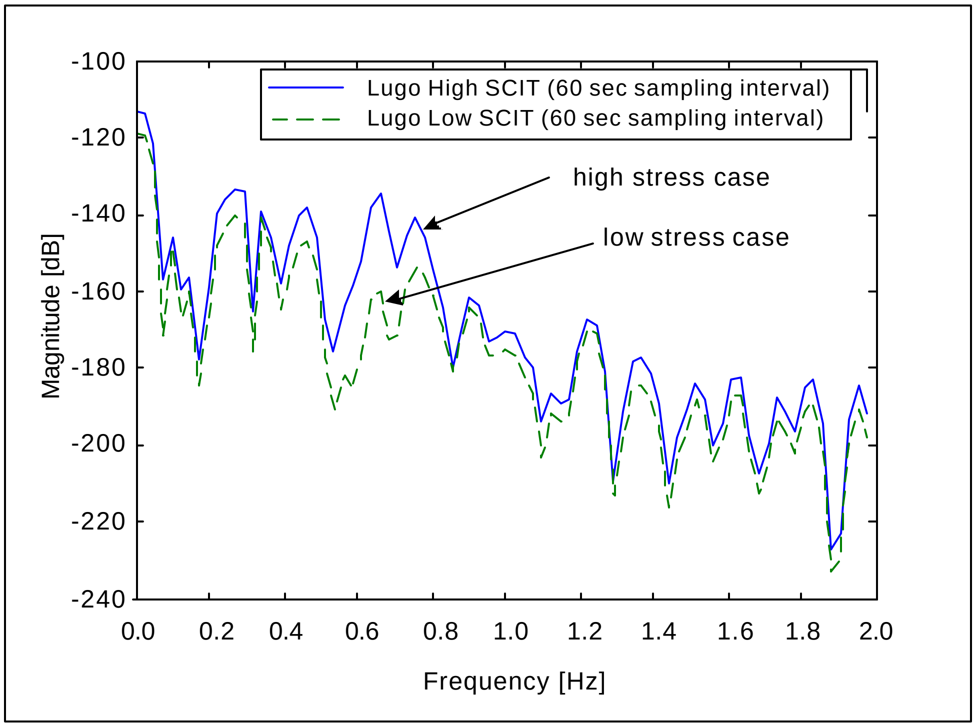

Figure 5-5: Autospectrum of System Frequency at Lugo, California. Generated by FFT with 60 second samples, Hanning squared window, and second-order low-pass filter breaking at 5 Hz (see Figure 5-3).

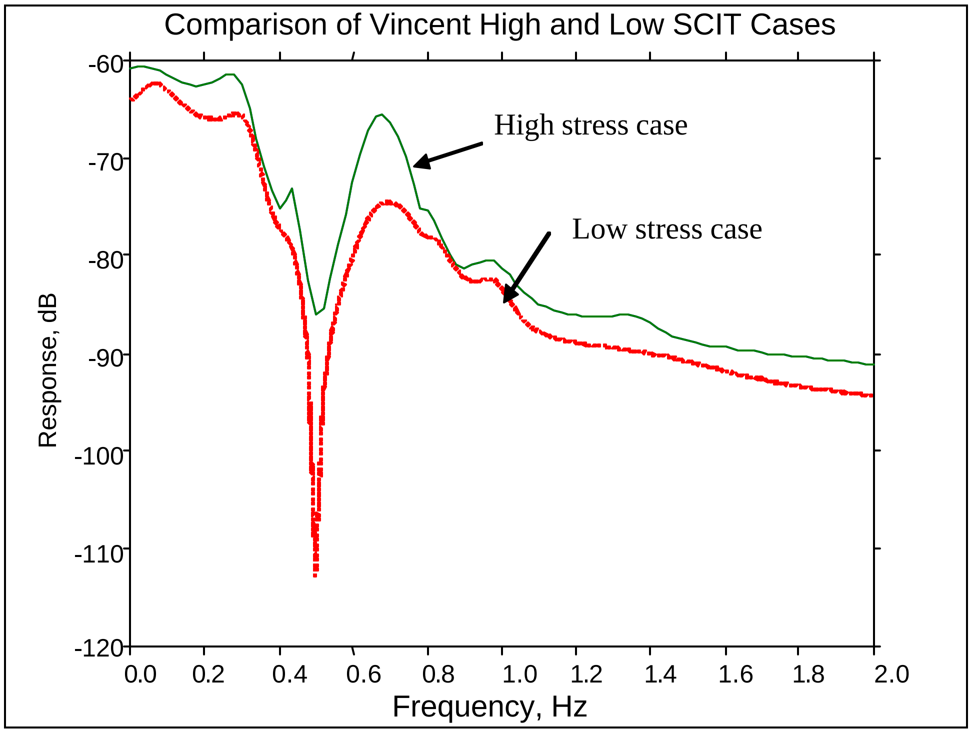

Figure 5-6: Response at Vincent, California, to Chief Joseph Brake event.

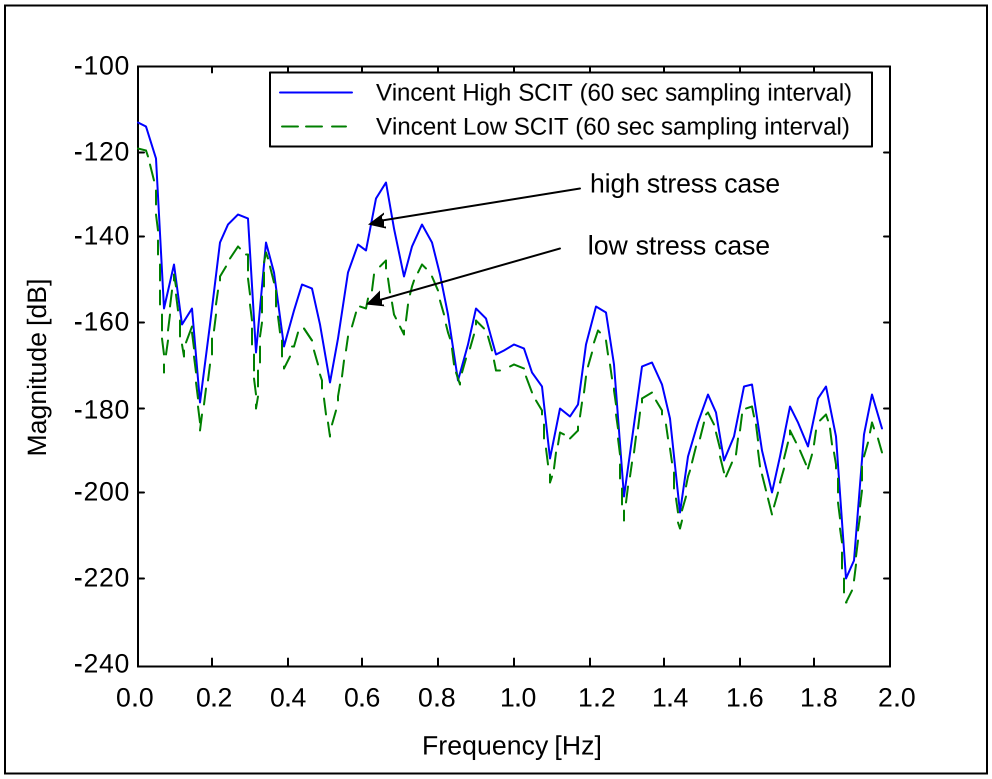

Figure 5-7: Autospectrum of System Frequency at Vincent, California. Generated by FFT with 60 second samples, Hanning squared window, and second-order low-pass filter breaking at 5 Hz.

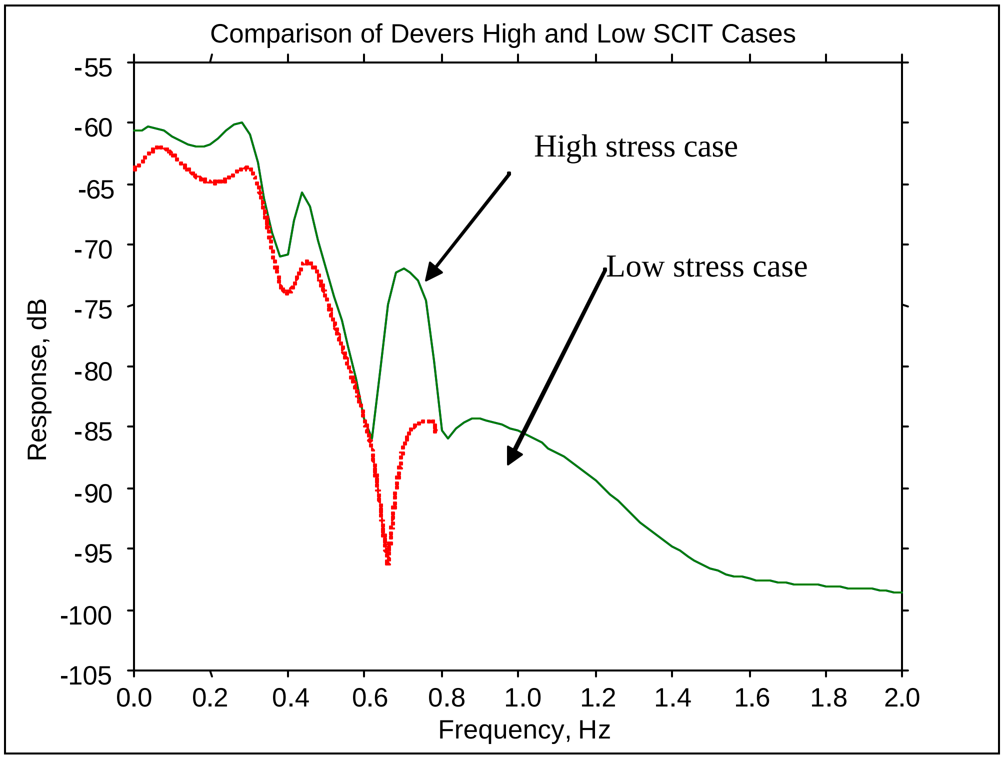

Figure 5-8: Response at Devers, California, to Chief Joseph Brake event.

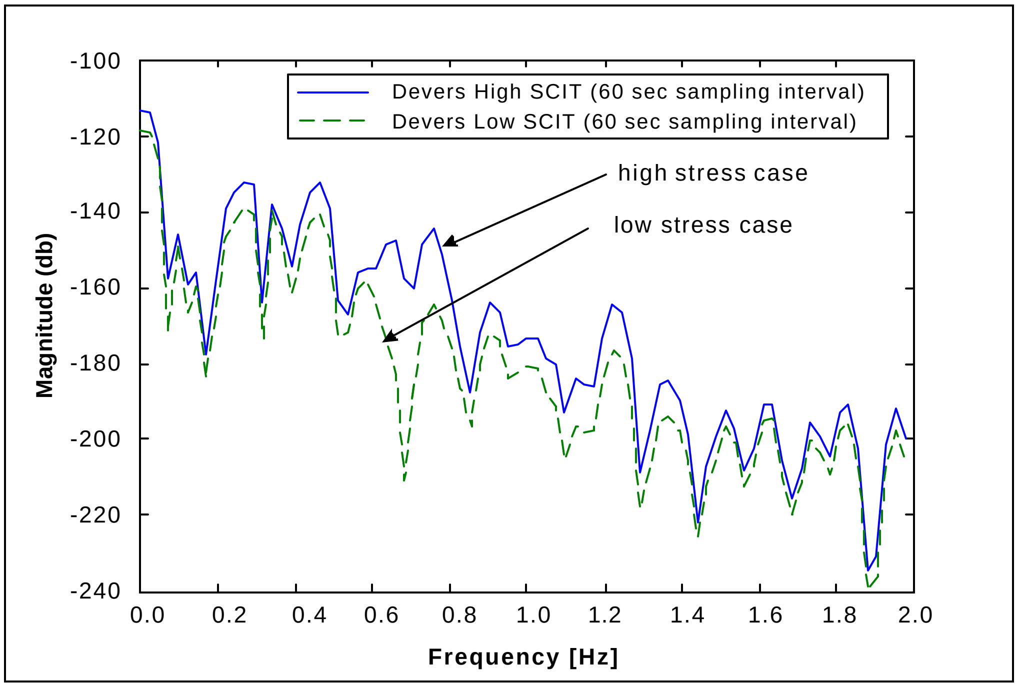

Figure 5-9: Autospectrum of System Frequency at Devers, California. Generated by FFT with 60 second samples, Hanning squared window, and second- order low-pass filter breaking at 5 Hz.

### 5.2.2.3 Real Data from the WECC Breakup of August 10, 1996

On August 10, 1996, the WECC experienced a system-wide breakup with major regional power outages. Figure 5-10 shows the autospectrum for the real power transient leading up to the separation of the interconnected system. The first of a series of events was the Keeler-Allston line trip. (See Figure 5-10 at about 400 seconds on the time axis). Figure 5-11 shows the autospectrum of the system frequency at the Dittmer Control Center in Vancouver, WA, before and after the line trip. The autospectra in Figure 5-11 have very similar characteristics to the simulated results shown in Figure 5-5, Figure 5-7, and Figure 5-9. A comparison between the low and high stress conditions and the before- and after-line-break conditions indicates that the spectrum of the high stress condition correlates with that of the after-the-line-break condition and, similarly, the spectrum of the low stress condition with that of the before-the-line break condition. This is a very intuitive result. The grid stress after the line break was likely significantly increased, having led 5 minutes later to a cascading effect, which ultimately caused a system breakup.

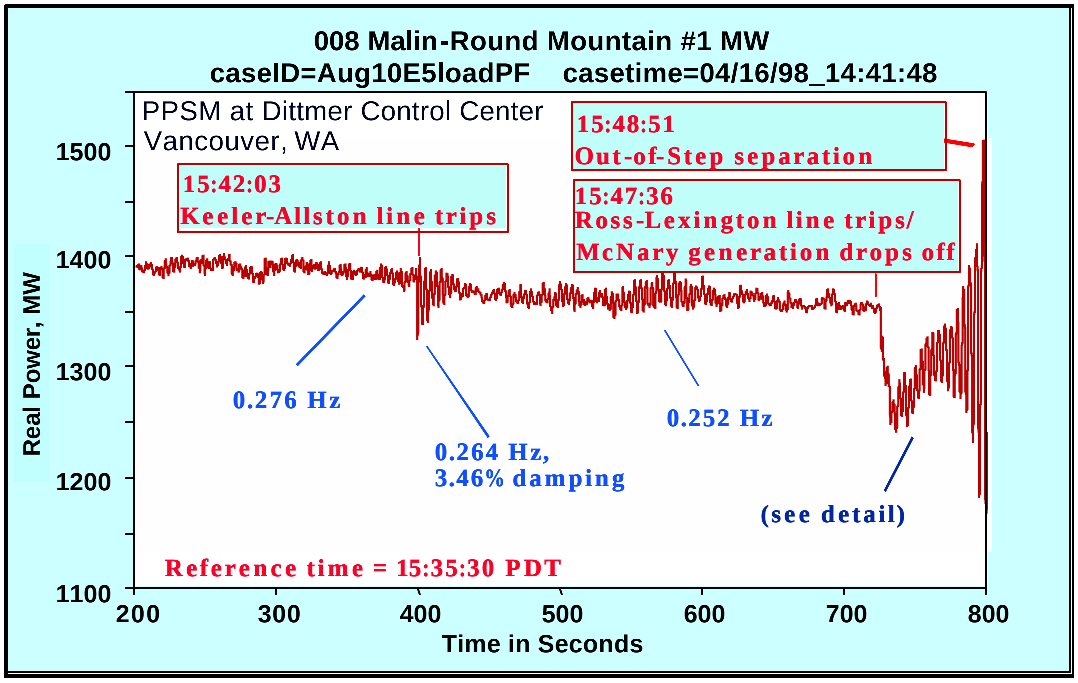

Figure 5-10: WECC Breakup of August 10, 1996. Shown is the real power at Malin. Several events leading to the separation of the interconnected power system are indicated.

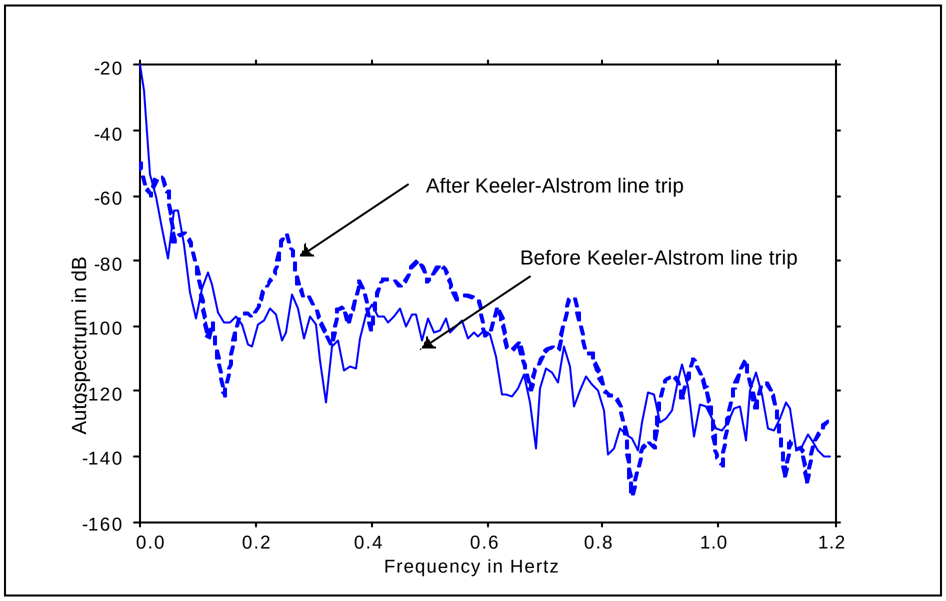

Figure 5-11: Spectrum of System Frequency Before and After the Keeler-Alstrom Line Break, Recorded at Dittmer Control Station, WA

### 5.2.3 Findings

The results of this analysis reveal the following:

- The simulation results indicated recognizable differences between the high and low stress cases in the frequency response of the system to the Chief Joseph Brake for different locations throughout California. The differences were all consistent.
- Recorded data from the WECC breakup of August 10, 1996, indicate similar characteristics before and after the Keeler-Allston line trip compared to the low and high stress cases in the simulation. It was shown that the oscillatory content in the real power after the Keeler-Alston line break increased, which then precipitated other contingencies, which led ultimately to the system breakup [Hauer 2001] (see Figure 5-10). It can be argued that grid stress after the Keeler-Alston line trip increased to a degree potentially detectable by an algorithm that would be implemented in the load controller. The Keeler-Alston line trip and its impact on the stability of the grid could be considered as an early warning sign, which the load controller will need to be able to detect.

The simulation results and the analysis of the recorded data for August 10, 1996, support our notion that in the high voltage transmission system, the system frequency signal contains important information as the power system changes from a low stress to a high stress state that could be exploited in a detection algorithm. While our analysis to date has only investigated frequency signals of three California locations, we expect that these results can be replicated for other locations in California and throughout the WECC area.

It should also be mentioned that the particular definition of grid stress used in this project is not a rigorous definition, but serves well to define impending grid related problems. There are potentially an infinite number of stress conditions that may be locationally dependent, however, most are low probability events.

### 5.2.4 Conclusions from Analysis of Simulation of Dynamic Stability

The results of the analysis to date support our basic hypothesis that local detection of global system problems is feasible. While we have demonstrated this concept for three locations in California, we feel optimistic that these results apply more generally for other locations in California and in other states of the WECC. Validating the hypothesis can be done by analyzing simulation outputs at different locations.

We also need to address the applicability of the results to the wall outlets in homes and businesses throughout California where ultimately the grid stress detection will need to occur. An important test would be to verify that the coherence of the frequency signals on each side of the substation transformer is unity. In the absence of sufficient dynamic data for the distribution feeder, we argue that there are no apparent reasons why the dynamic content of the frequency signal should significantly change downstream of the substation in the distribution feeder leading to homes and other buildings. With no generation capacity or electric storage capability, it is unlikely that loads will induce oscillatory behaviors into the distribution system. Frequency, however, is by definition the time derivative of the voltage phase angle, therefore, the instantaneous frequency measurement is influenced by sharp changes in voltage, which can occur within distributions systems. Further signal processing could be used to account for these errors.

In summary, we have generated encouraging results that warrant and suggest following the next steps toward the design of a detection algorithm. With the observed geographic diversity in the frequency spectra results, it is likely that a successful detection algorithm would need to be adaptive to adjust to regional differences. The algorithm would need to continuously establish a spectral baseline that is specific for a particular location, similar to Kahlman filtering methods. The detection of significant changes from that baseline could then be used as a trigger for load control.

## 5.3 Voltage Stability Conditions

In discussion with CAISO transmission planning engineers, CAISO engineers indicated that the voltage instability in the Bay Area during heavy AC/DC North-to-South power flows is of great concern to CAISO and suggested that this particular case be analyzed using the simulation approach described in the previous section.

CAISO staff provided the necessary input data sets for the PSLF simulation program and consulted with us on the definition of low- and high-stress conditions using the T-116B nomogram [CAISO 2002]. This nomogram indicates safe operation of the grid as a function of PDCI (Pacific Direct Current Intertie) flow, COI (California Oregon Intertie) flow, and Northern California hydroelectric generation dispatched. The nomogram protects against voltage instability at the Table Mountain Substation for specific transmission outages described in CASIO operating procedure T-116 (see Figure 5-1).

### 5.3.1 Approach

The same analytic approach as for dynamic instability conditions was applied. The steps are:

1. Using the PSLF simulation program, we varied the Northern California hydroelectric dispatch between 100% and 70%. We defined the 100% hydroelectric dispatch as the high-stress case and 70% dispatch as the low-stress case. We consulted with CAISO engineers for specific plant dispatches for both simulations.

2. For steady-state cases of Northern California Hydro = 100% and 70%, dynamic simulations were performed using a 0.5 second Chief Joseph brake ringdown. Voltage signals at the high voltage bus for Table Mountain, Round Mountain, and Vaca Dixon, were captured during the ringdown, and converted to transfer functions using Prony analysis, thus providing a transfer function for each site $G_{pv}(s)$. $G_{pv}(s)$ is a transfer function with real power as input and voltage as output.

3. Gaussian white-noise inputs were filtered (breakpoint at 5 Hz), then input into each transfer function (see Figure 5-3) . An FFT was performed on the output, and differences among signals between 100% and 70% hydro dispatch levels were scrutinized for signatures indicating stress of impending voltage instability.

### 5.3.2 Findings and Conclusions

The transfer functions $G_{pf}(s)$ of the entire power system at Table Mountain indicated virtually no differences between the low- and high-stress cases for frequencies below 0.8 Hz, and some differences for frequencies above 0.8 Hz (see Figure 5-12). The spectra of these transfer functions, when imposed with a noise signal, are identical for the two stress cases (see Figure 5-13). Similar results were obtained for the Round Mountain Substation (see Figure 5-14 and Figure 5-15). This leads to the conclusion that an impending voltage instability problem may not be detectible with dynamic analysis techniques as we postulated in the previous section. The fundamental principals of a voltage instability problem in a complex power network are therefore, not associated with the dynamic systems behavior. While our simulation validated this notion, there is still a debate within the power system engineering community regarding whether evidence exists for impending voltage instability in a dynamic simulation that reveals oscillatory behavior. Oscillations in the WECC, which generally occur in the 0 to 2 Hz region, are primarily of electro-mechanical nature, thus they are typically not directly associated with voltage instability, although sometimes they are observed.
We conclude from our simulation results that the proposed grid-stress detection approach appeared inappropriate for detecting voltage instability problems; however, the approach is likely to have merits for dynamic instability problems, as shown in Section 5.2.

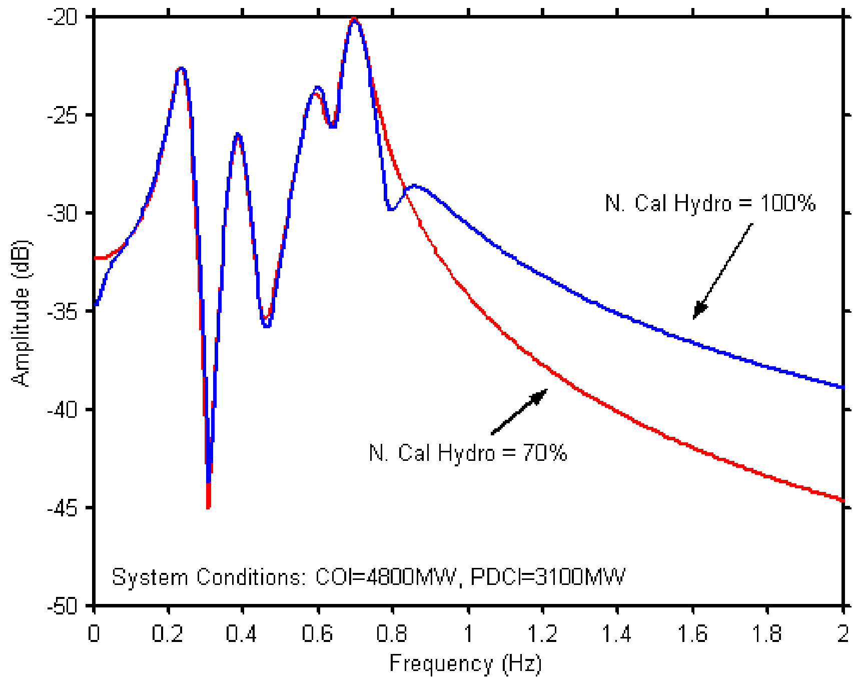

Figure 5-12: Transfer Function $G_{pf}(s)$ at Table Mountain, California, for High-Stress Case (North California Hydro=100%) and Low-Stress Case (North California Hydro=70%).

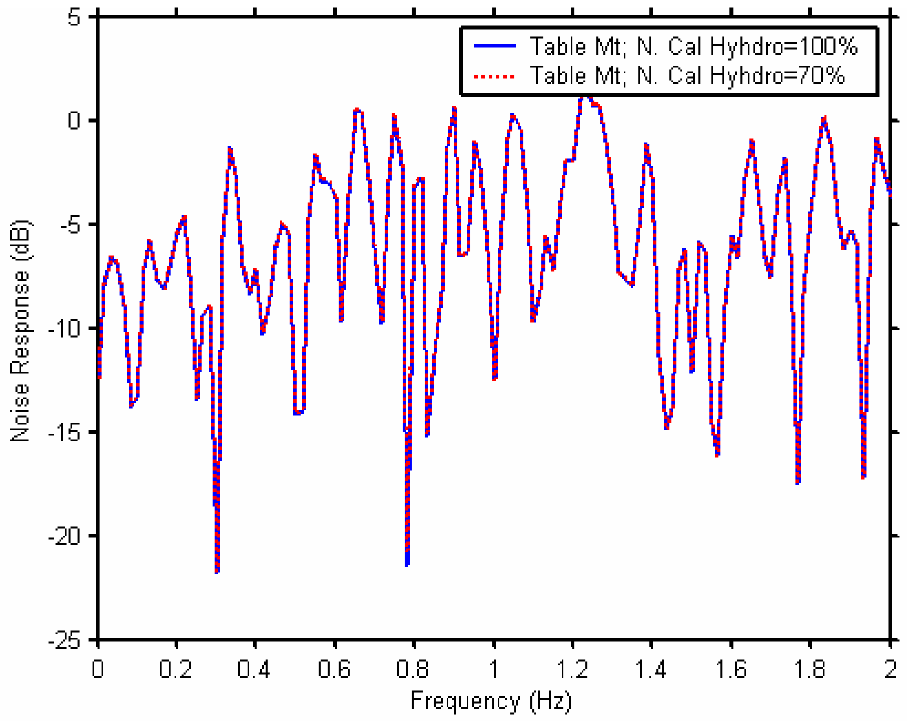

Figure 5-13: Spectrum of System Frequency at Table Mountain, California, for High-Stress (North California Hydro=100%) and Low-Stress (North California Hydro=70%). These results were generated by FFT with 60 second samples, Hanning squared window, and 2nd order low-pass filter breaking at 5 Hz (see Figure 5-3). Note that both spectra are identical.

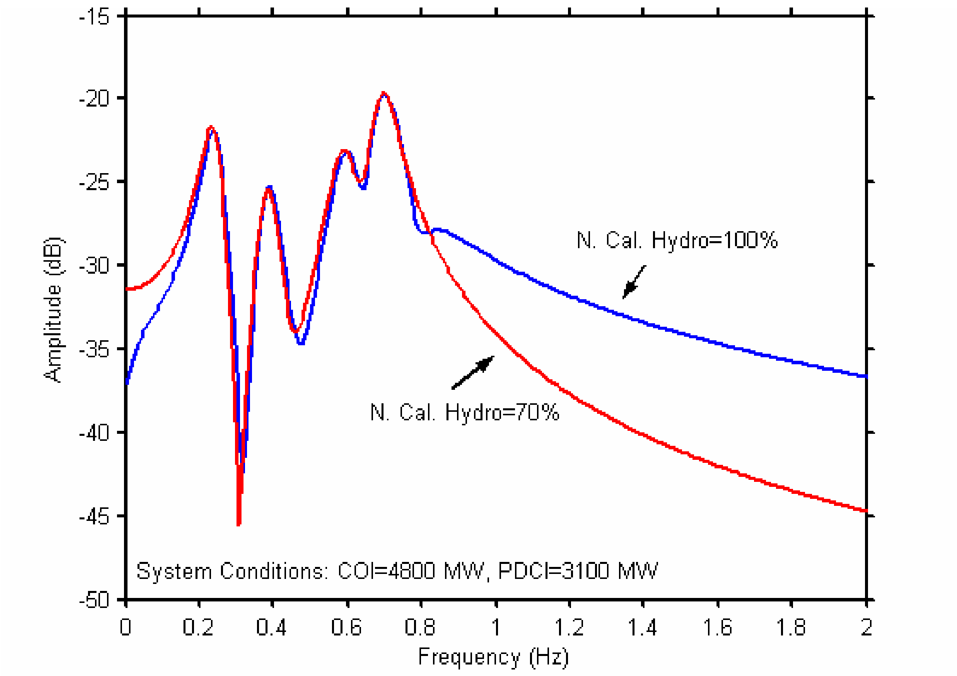

Figure 5-14: Transfer Function $G_{pf}(s)$ at Round Mountain, California, for High-Stress Case (North California Hydro=100%) and Low-Stress (North California Hydro=70%).

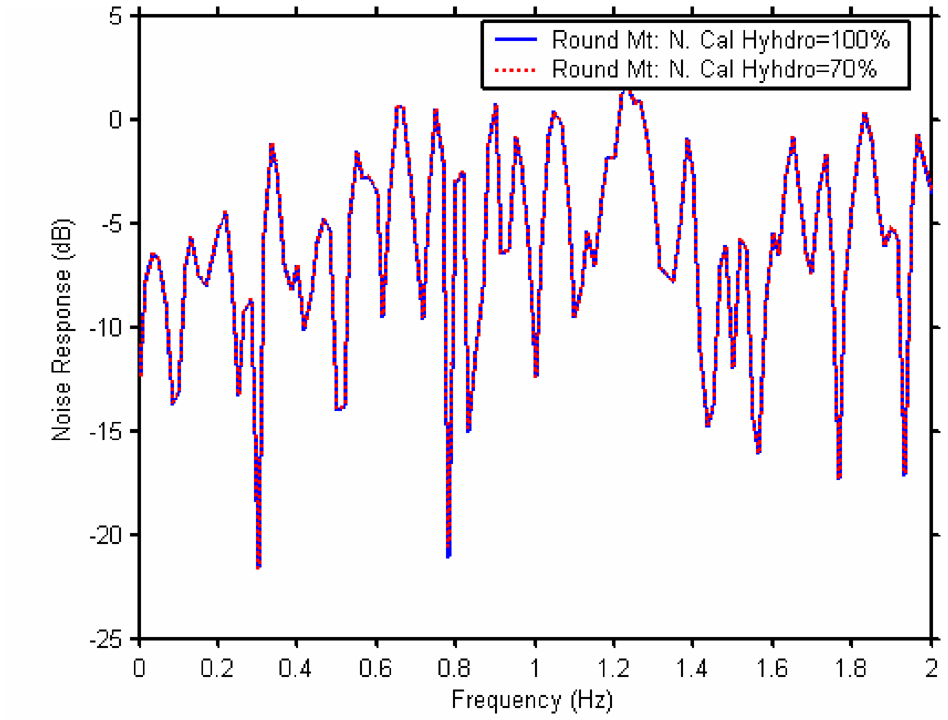

Figure 5-15: Spectrum of System Frequency at Round Mountain, California, for High-Stress (North California Hydro=100%) and Low-Stress (North California Hydro=70%) cases. Results were generated by FFT with 60 second samples, Hanning squared window, and 2nd order low-pass filter breaking at 5 Hz (see Figure 5-3). Note that both spectra are identical.

# 6 Analysis of Grid Events

Using the analysis tools described in Section 5, we analyzed historic data of known grid events to test the hypothesis established in Section 4. Two grid events were analyzed: (1) the August 10, 1996, breakup of the WECC system that caused major outages in the Western power system and (2) a less severe event that was caused by tripping of a major transmission line in the Northwest with cascading effects in Southern California.

Simulation results suggested that some evidence exists for the maxima of the spectrum of the grid frequency signal tending to be higher in the 0 to 2 Hz range for the high-stress condition compared to a low-stress condition. Furthermore, we found that there is also evidence that the maxima are sharper (spanning a small angle) in the high-stress case versus the low-stress case. There appears to be sufficiently significant differences in the spectra of grid frequency signals between the two cases to potentially establish detection algorithms based on them. The differences are expected to be found in the sharpness of the spectrum and in the absolute magnitude of the spectrum. We tested these two characteristics by analyzing the spectra using following characteristics:

- Integral of spectrum in each band
- Number of occurrences of peaks in specific frequency bands
- Standard deviation of the spectrum.

## 6.1 Results for the October 8, 2002, Disturbance

On October 8, 2002, at 22:30:52 Pacific Daylight Saving Time, a line trip in southern Oregon near Summer Lake caused Bonneville Power Administration to execute remedial action schemes (RAS) to trip 2908 MW of generation in the Northwest. Additionally, the 1400 MW Chief Joseph Brake was immediately inserted. Load losses occurred at scattered locations, and various local control actions may have also occurred.

### 6.1.1 Time Domain

As a result of the line trip, the grid frequency decreased significantly and very fast (see Figure 6-1). The recovery time to a normal grid frequency of 60 Hz was about 1000 seconds (17 minutes). The lowest frequency measured occurred at 22:30:57 and 57 ms and was 59.593 Hz. This is far outside the tolerance range of ±0.05 Hz (about 8 times more) and occurred only for a very short time. The average frequency after the trip was about 59.78 Hz for about 3 minutes. The frequency then started to increase with a slope of about 0.22 mHz/sec. Finally, after approximately 1000 sec, the nominal frequency of 60 Hz was reached again.

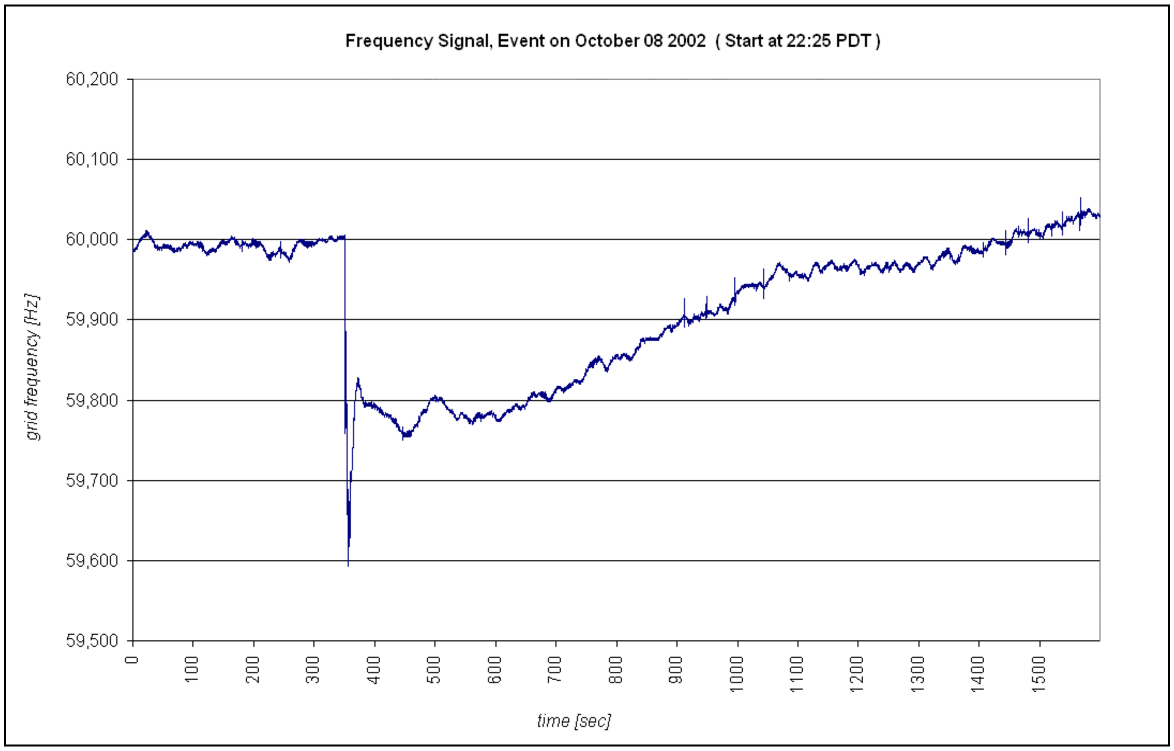

Figure 6-1: Grid Frequency Event, October 8, 2002.

### 6.1.2 Magnitude of Maxima in Spectrum

The first part of the analysis is based on the magnitude of the highest peaks in the frequency spectrum. This is an indication for high frequency components and lets us determine oscillatory modes in the power system.

Figure 6-2 shows the magnitude of the maxima for each band over a 10-minute time period. The line break is indicated in Figure 6-2 in the center of the graph by a vertical line. We noticed that the magnitude of the maxima in bands 1, 2 and 3 increased after the trip for about 1 minute. In band 4 and 5, no rise or other distinct characteristics in the trajectory of the peaks was detectable. This suggests that only lower frequency oscillations (up to 1.2 Hz) emanated from this line trip.

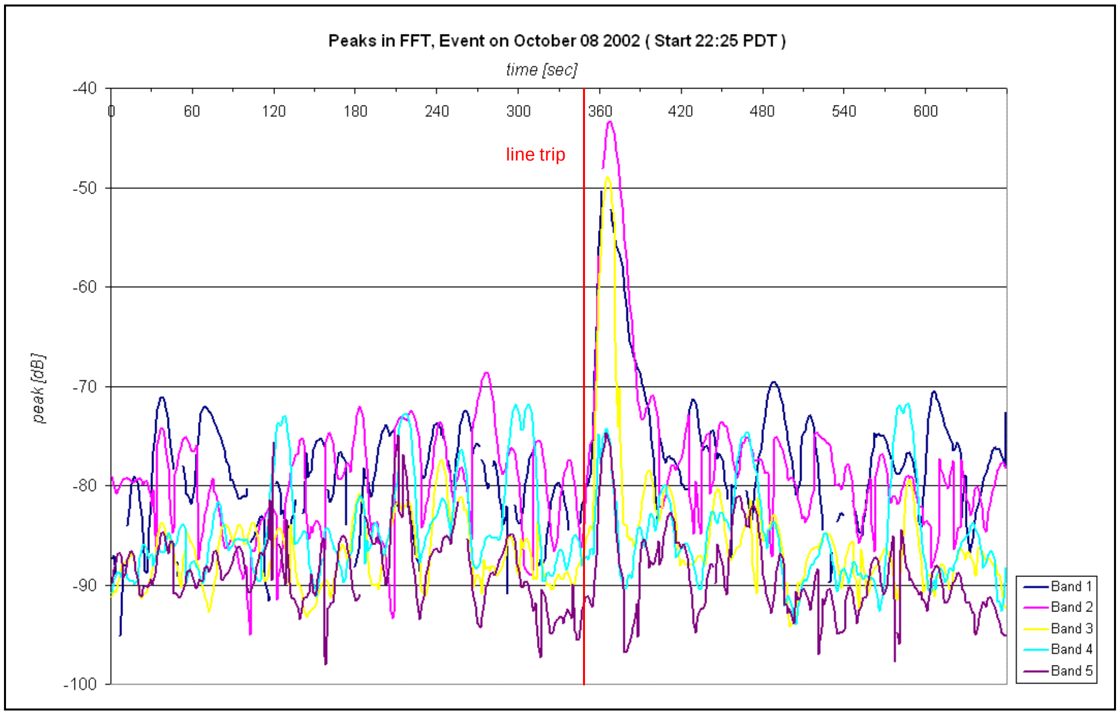

Figure 6-2: Magnitude of Peaks in Spectrum, October 8, 2002

### 6.1.3 Standard Deviation

Figure 6-3 shows how the standard deviation of the magnitude of the spectrum varies over time before, during, and after the event. The standard deviations are computed over the complete bandwidth between 0 and 2 Hz of the spectrum. After the trip, an increase in standard deviation is detectable for a short time, compared to the recovery time of the grid frequency to 60 Hz.

A high standard deviation in the magnitude of the spectrum is indicative of a wide spread between dominating and non-dominating frequencies during the first 1 minute after the line break.

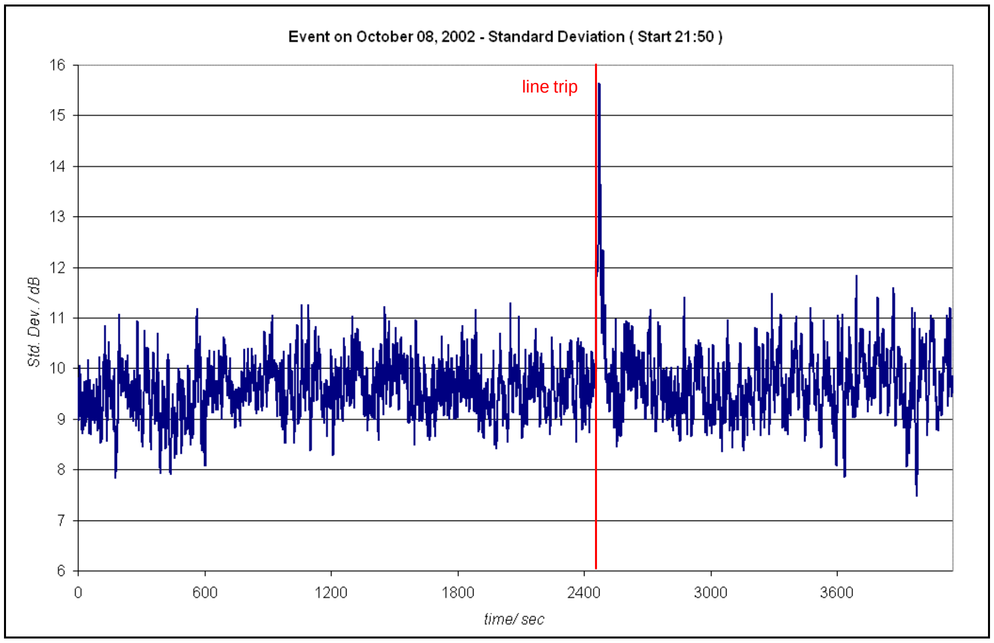

Figure 6-3: Standard Deviation of Magnitude of the Grid Frequency Spectrum, October 8, 2002

### 6.1.4 Sharpness of Maxima

Figure 6-4 illustrates an example of the trajectory of the second derivative for Band 2. Trajectories of the second derivative for other bands were very similar to that shown in Figure 6-4 with no characteristic feature at the time of the line trip or thereafter.

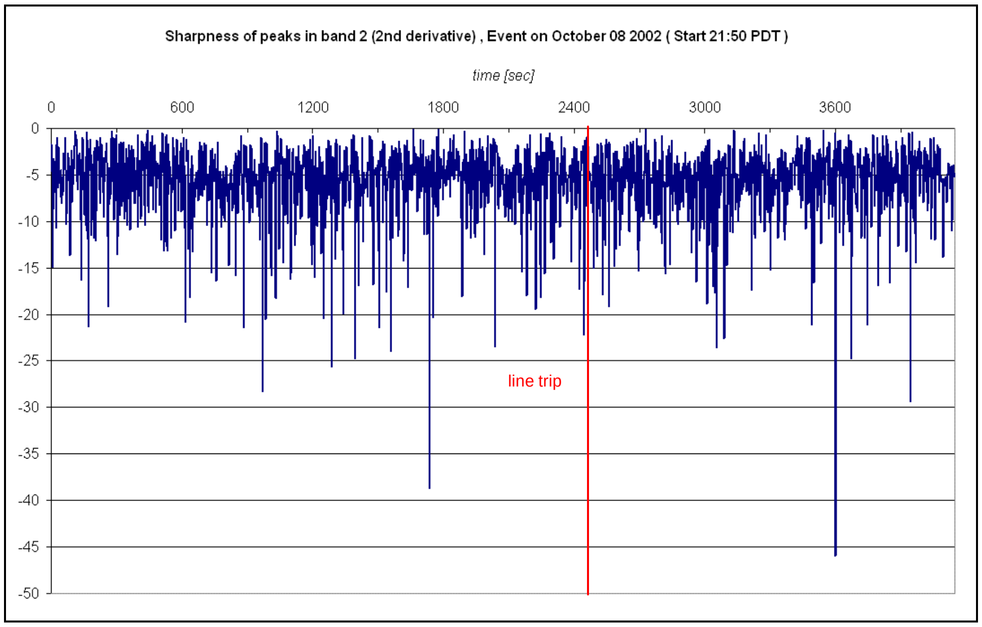

Figure 6-4: Sharpness as Defined by the Second Derivative of Spectrum for Band 2, October 8, 2002

The alternative method for determining sharpness, utilizing an angle measurement, provides the results shown in Figure 6-5. Results shown in Figure 6-5 are valid for Band 2 only. They are in strong agreement with the results using the second derivative approach.

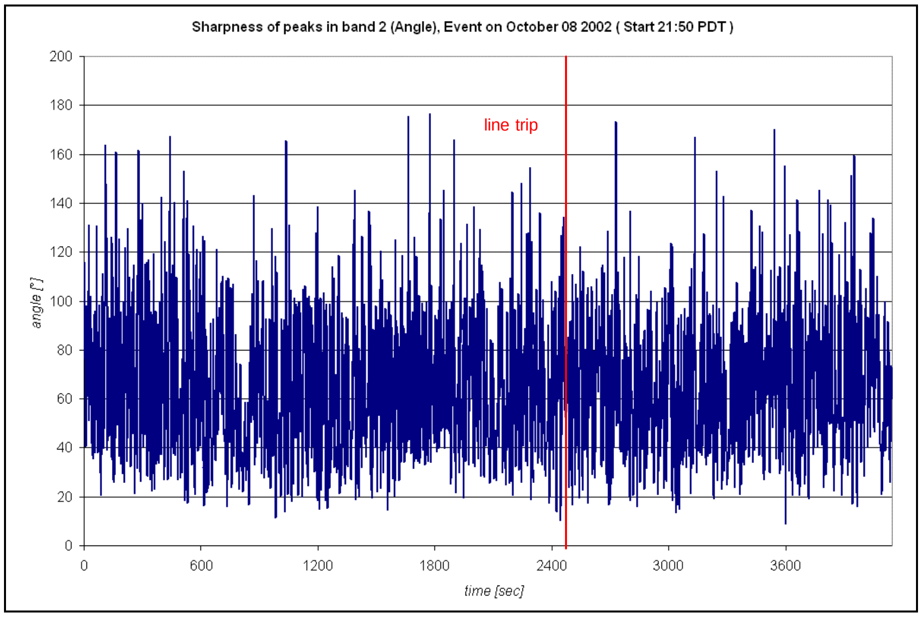

Figure 6-5: Sharpness as Defined by Angle at Maximum of Spectrum, October 8, 2002

### 6.1.5 Integral

The integral is separately calculated for each band. It represents the surface area between an arbitrary reference line of –120 dB and the line representing the spectrum. Figure 6-6 shows results for the integral for the Band 2 spectrum. There is a markedly high rise in the value of the integral directly after the line trip, which persists for only a short duration of 8 seconds. Similar results were obtained for Bands 1 and 3, which indicates that the oscillatory content of the frequency signal as a consequence of this grid event is limited to frequencies up to 1.2 Hz. Oscillations at higher frequencies beyond 1.2 Hz were not observed. This observation is in strong agreement with known oscillatory modes, ranging from the low Canada-California mode at 0.33 Hz to the higher frequency Grand Coulee mode near 1.03 Hz [Hauer and Dagle 1999]. The system's damping ability was sufficiently large to arrest persistent oscillations within a very brief period.

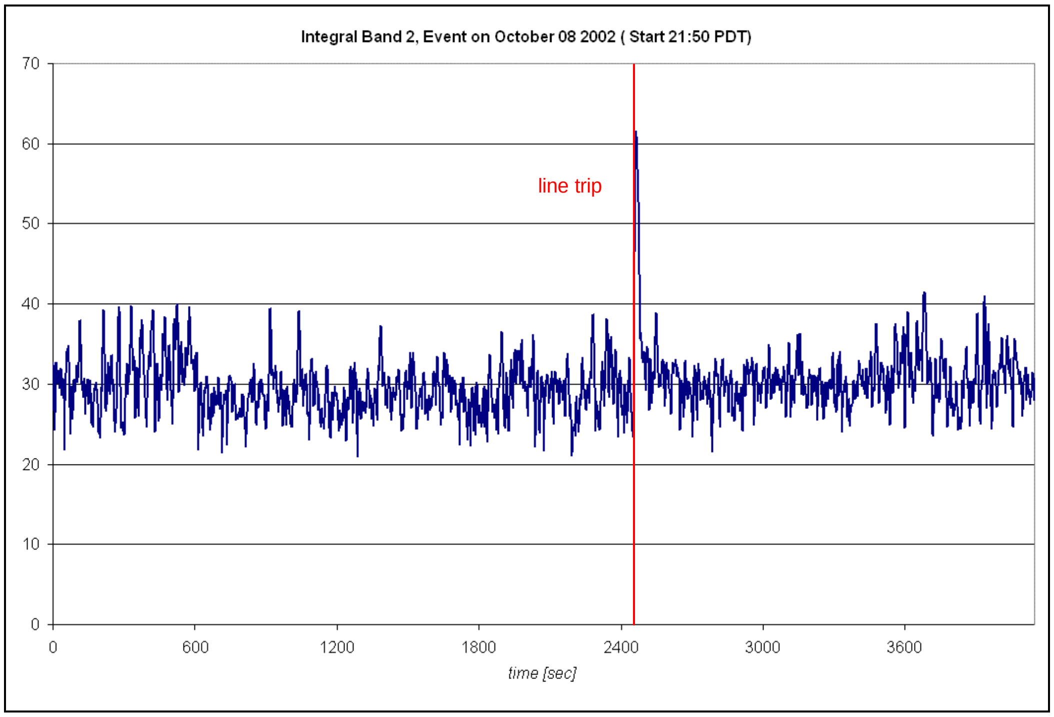

Figure 6-6: Value of Integral between -120 dB Reference Line and Spectrum for Band 2, October 8, 2002

### 6.1.6 Histogram of Maxima

The frequency of occurrence of local maxima before and after the line trip was explored to investigate any modal changes in the oscillation prior to and after the line break. If there was a change in the mode of oscillation because of the line trip then we could expect a change in the distribution of the local maxima before and after the event. The histogram was normalized such that all distributions summed to 100%. Thirty-minute periods prior to and after the event were analyzed and the results compared.

Figure 6-7 shows histograms for each band before and after the line trip. The frequencies at which the spectra peak before and after this event differ only very slightly. This means that there has not been any shift in the dominant oscillatory behavior from before to after the event. Sometime during major grid events the grid topology changes as a result of the line break and significant load shedding. This affects the oscillatory behavior of the entire western interconnected system. Figure 6-7, however, suggests that this is not the case. There does not appear to be major topological changes resulting from the line break causing a shift in the oscillation modes of the system.

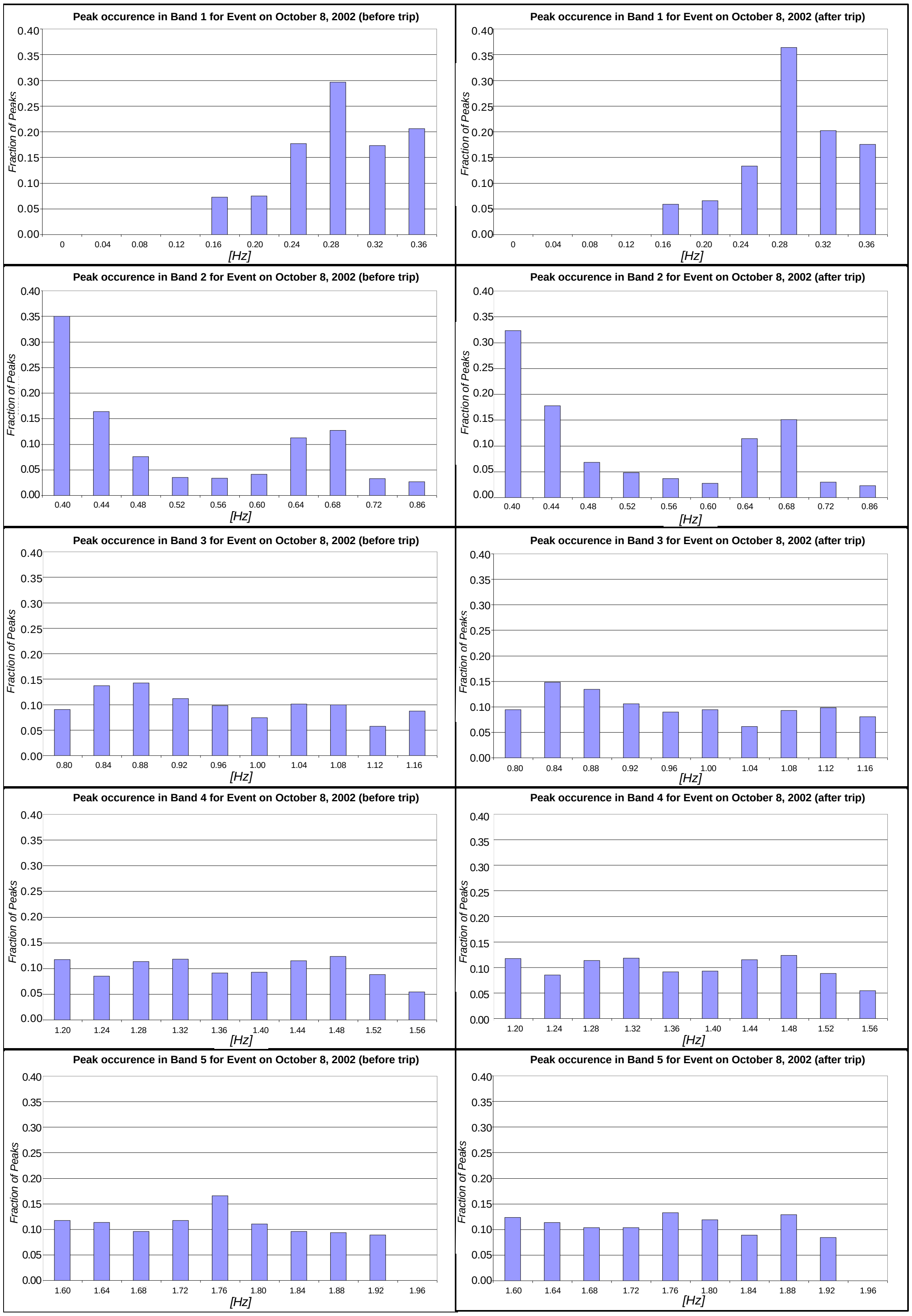

Figure 6-7: Histograms of Local Maxima, October 8, 2002

## 6.2 Results for August 10, 1996

On August 10, 1996, the interconnected western grid separated into islands and caused wide spread outages throughout the western US and Canada [Hauer and Dagle 1999]. The grid separation occurred in a succession of several events over a period of less than 10 minutes. First, the Keeler-Allston transmission line tripped, followed by the Ross-Lexington transmission-line trip and, almost concurrently, the tripping of the McNary generator, located on the Columbia River. The system separation occurred after the McNary generator trip, causing widespread outages throughout the western US for several hours.

In comparison to the previously discussed October 8, 2002, event, this event is divided into three different phases:

- Phase 1: Before the Keeler-Allston line trips
- Phase 2: Between the Keeler-Allston line trips and Ross-Lexington line trips
- Phase 3: After the Ross-Lexington line trips.

The focus of this analysis is to detect significant changes in the set of five criteria between Phases 1 and 2. Phase 1 is presumed to be an example for low stress, and Phase 2 is presumed to be characteristic of high stress. Phase 3 finally represents the case of instability.

Figure 6-8 illustrates the power transfer from the Northwest to California through the Malin-Round Mountain transmission line (see Figure 5-1) around the time of the critical events. The instability of the system after the second event can be recognized by the undamped oscillation of the power output.

Similar oscillations, however not as pronounced, are shown in the grid frequency as illustrated in Figure 6-8.

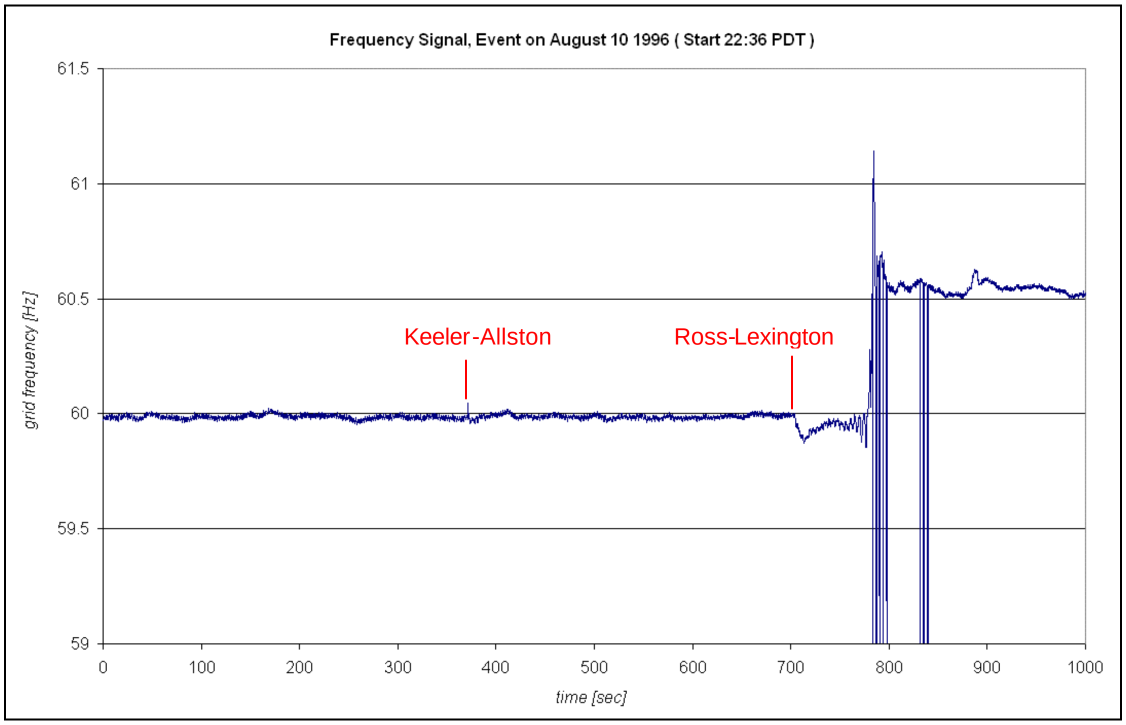

Figure 6-8: Grid Frequency during WECC Breakup of August 10, 1996

A minor increase in the frequency to 60.045 Hz over a 1-second period was recorded by the Keeler-Allston line trip. The Ross-Lexington line trip (second event) caused grid frequency to drop to about 59.88 Hz with ensuing undamped oscillations, which ultimately led to the breakup of the interconnected system into numerous islands.

### 6.2.1 Magnitude of Maxima in Spectrum

Figure 6-9 shows the magnitude of maxima in the spectrum for each of the 5 bands. The results for Band 1 indicate the largest increase after the first event (Keeler-Allston line trip).

Band 1 shows the highest increase after the Keeler-Allston line trips, with some persistence during the second phase. Other bands provide no marked differences between before and after the Keeler-Allston line break.

During the third phase (after the Ross-Lexington line trip), the magnitude of maxima in all bands indicate high oscillatory content in the grid frequency. A detection of impending grid problems at that stage, however, would most likely be too late to avoid catastrophic system failure. It would be desirable to detect the impending problem at an earlier stage, for instance after the Keeler-Allston line trip to prevent ensuing cascading effects.

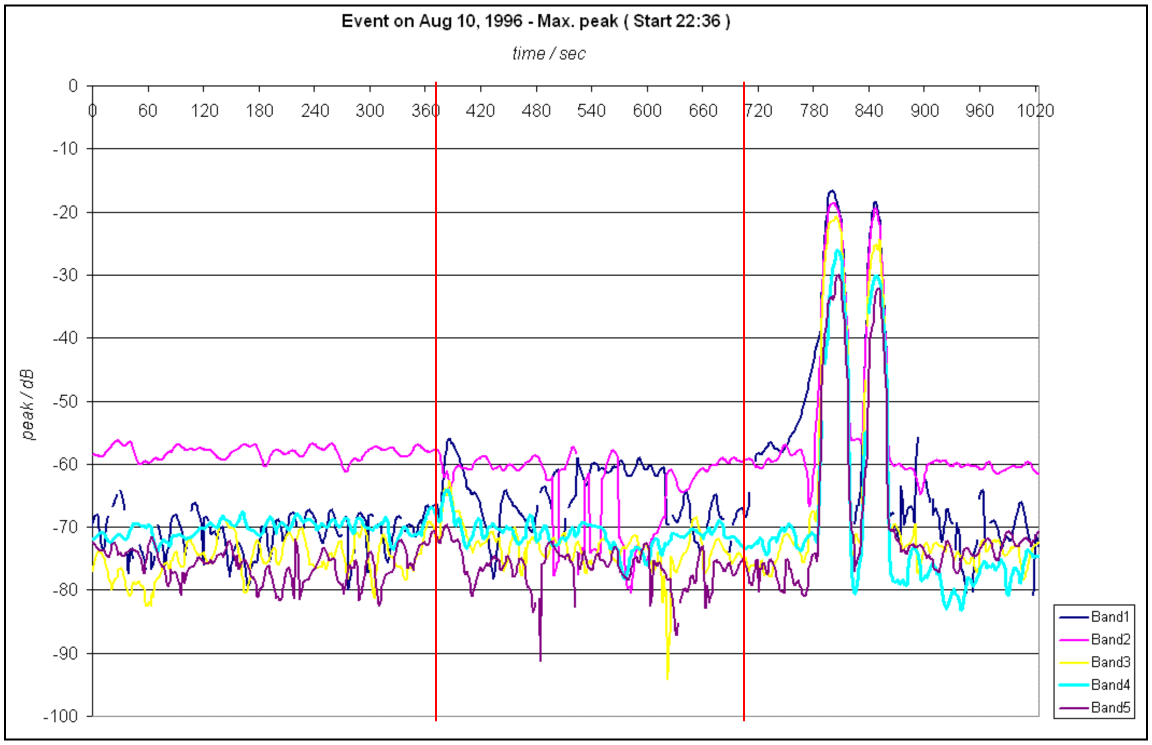

Figure 6-9: Magnitude of Maxima of Spectrum for Five Frequency Bands, August 10, 1996

### 6.2.2 Standard Deviation

The standard deviation of local maxima of the spectrum, as shown in Figure 6-10, does not reveal any significant characteristics transitioning from Phase 1 into Phase 2. Similar to the trajectory of the magnitude of maxima, significant changes in the characteristics emerge after the Ross-Lexington line break on the way to the total system collapse.

As a consequence, the standard deviation does not reveal any more information than that already obtained from the trajectory of the magnitude of maxima.

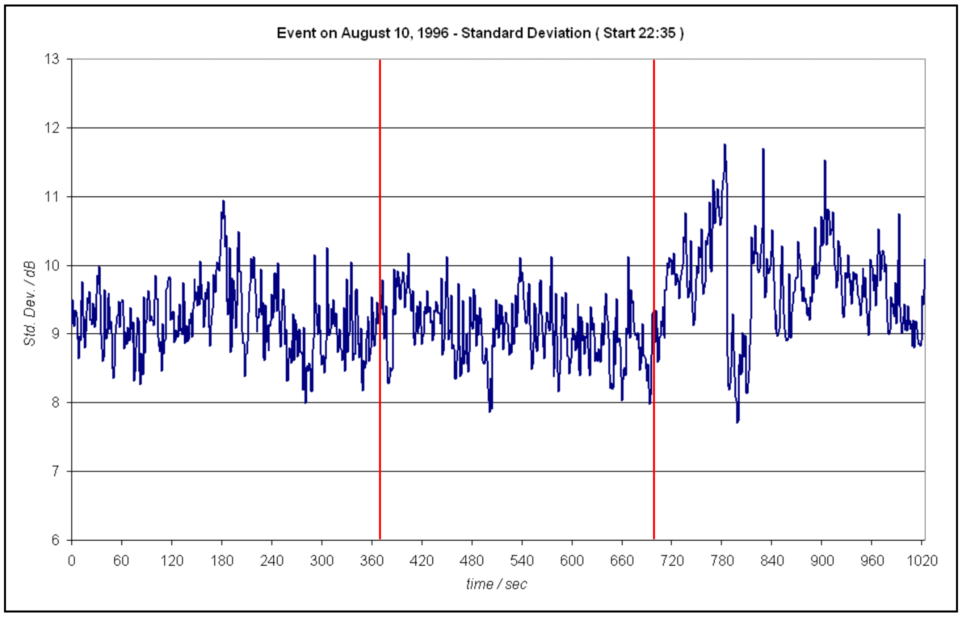

Figure 6-10: Standard Deviation of Local Maxima, August 10, 1996

### 6.2.3 Sharpness of Maxima

The sharpness indicators either computed by the second derivative of the magnitude of the spectrum or by the angle at local maxima of the spectrum showed no significant changes across all 5 frequency bands. In fact, comparing the sharpness indicators of Phase 1 with those of Phase 2, showed a behavior opposite to the behavior postulated in our hypothesis. Figure 6-11 illustrates that the second derivative increases. Likewise, Figure 6-12 suggests the angle grows during transitioning from Phase 1 to Phase 2.

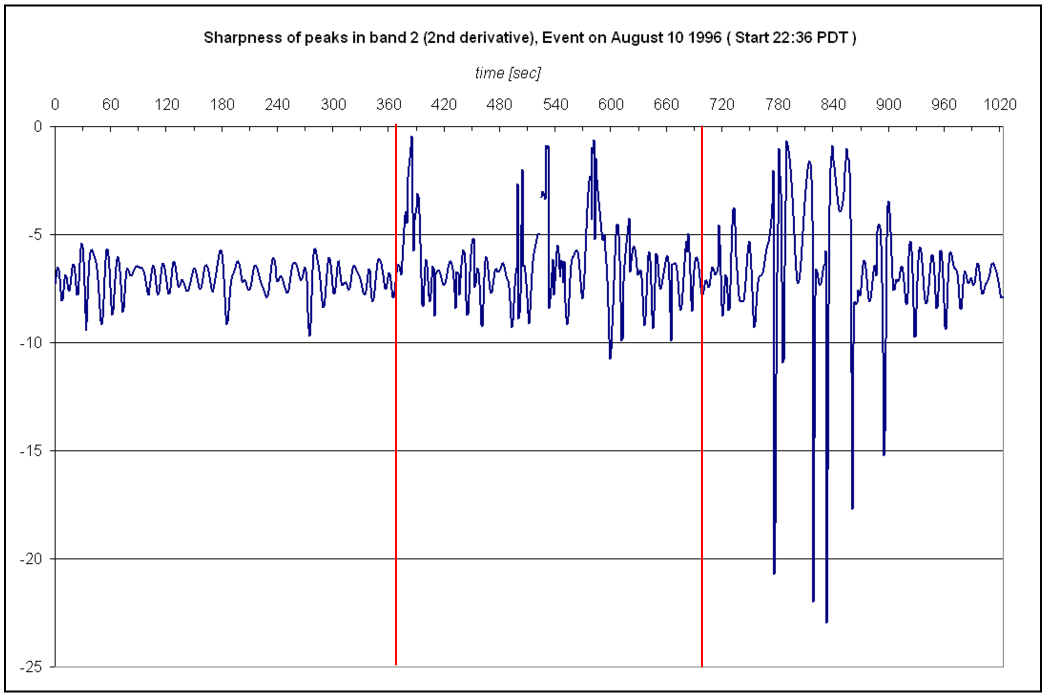

Figure 6-11: Sharpness as Defined by the 2nd Derivative of Spectrum for Band 2, August 10, 1996

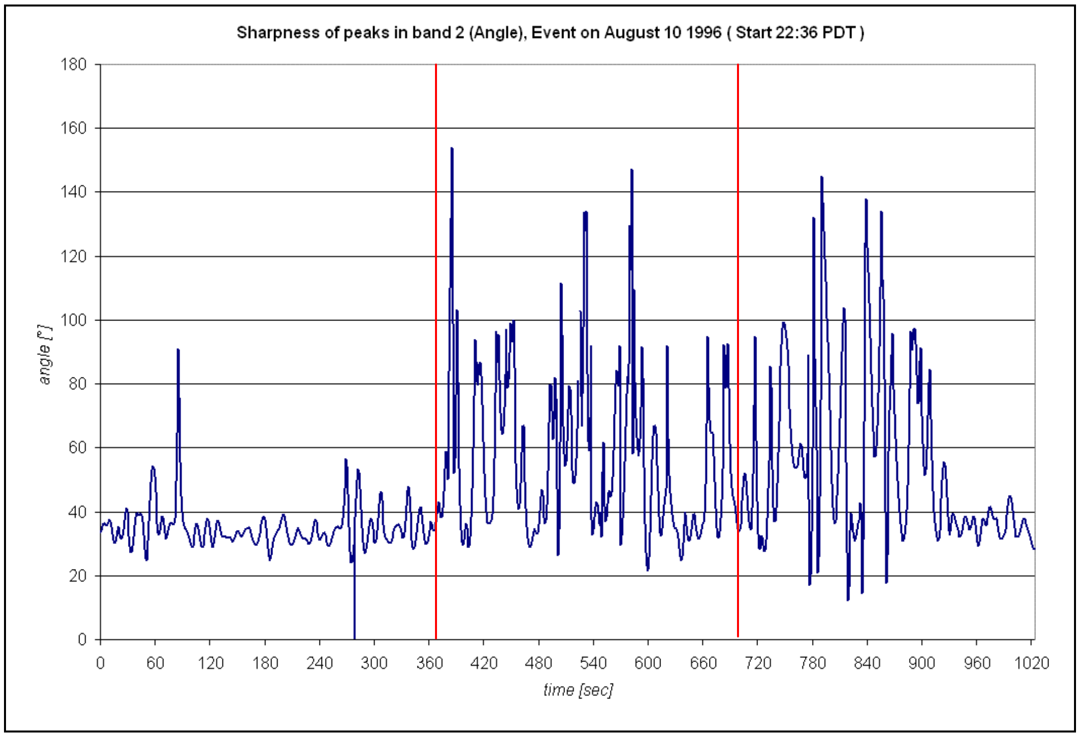

Figure 6-12: Sharpness as Defined by Angle at Maximum of Spectrum, August 10, 1996

### 6.2.4 Integral

There are two characteristics that are detectable in all five bands: 1) a small increase directly after the Keller-Allston line trips and 2) two large peaks after the Ross-Lexington line trips. This consistency is somewhat unique compared to the previously discussed indicators. It is still questionable whether the first rise in the indicator would be sufficiently unique to be utilized as a load-shredding signal.

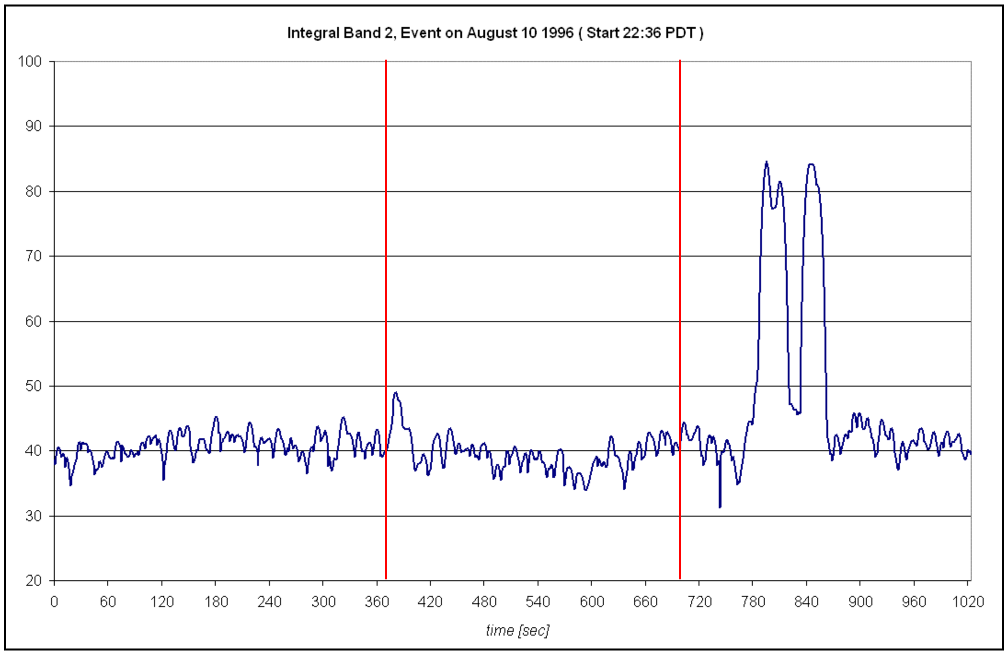

Figure 6-13: Value of Integral between -120 dB Reference Line and Spectrum for Band 2, for the August 10, 1996 Event

### 6.2.5 Histogram of Maxima

For all bands, noticeable changes in the histograms of the maxima were observed. They are indicative of significant changes in the oscillatory modes of the western power system and have been researched and reported in other work [Hauer and Dagle 1999]. One example is shown in Figure 6-14, Figure 6-15, and Figure 6-16, in which significant shifts in the major oscillation modes are noticeable. In Phase 1 (before the Keeler-Allston line break), a dominant oscillation of 0.45 Hz[5], generally referred to as the Alberta mode, was observed. After the Keeler-Allston line break, the primary oscillations were redistributed to higher and lower frequencies to reach a new dominant mode near 0.52 Hz in Phase 3.

[5] Figure 6-14 shows bins for the histogram in equidistant steps of 0.04 Hz. Because of the discrete binning method, all of the 0.45 Hz oscillation modes are accounted for in the 0.44 Hz bin.

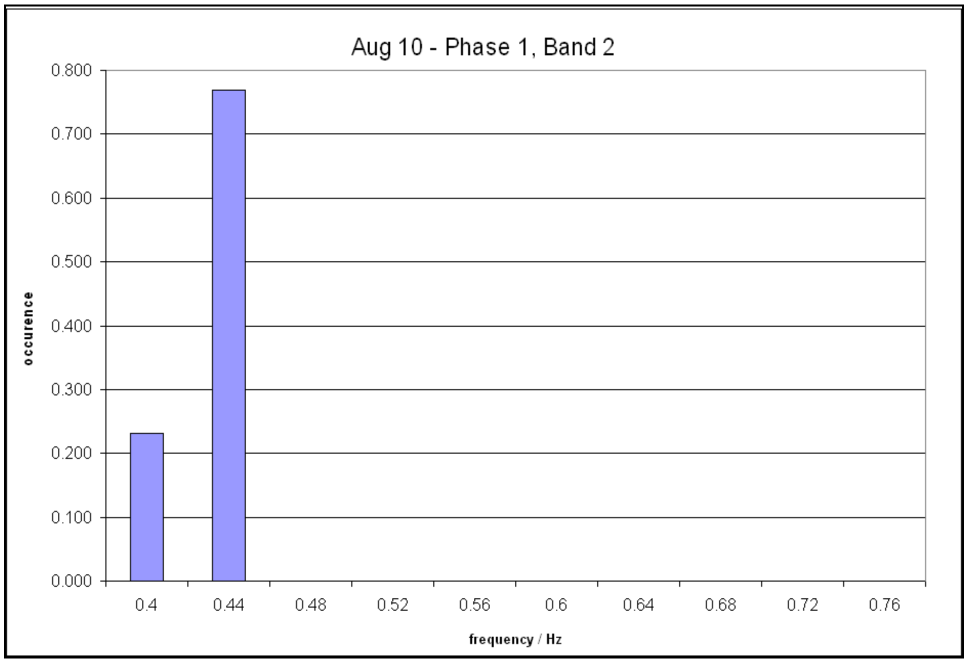

Figure 6-14: Histogram of Maxima, Band 2, Phase 1, August 10, 1996

It is difficult to interpret the results in isolation without other indicators. It was hoped that there were other confirming indicators, which were originally conceived when we postulated our hypothesis. In the absence of these results, we provide recommendations for alternative approaches, using the insights resulting from this research effort.

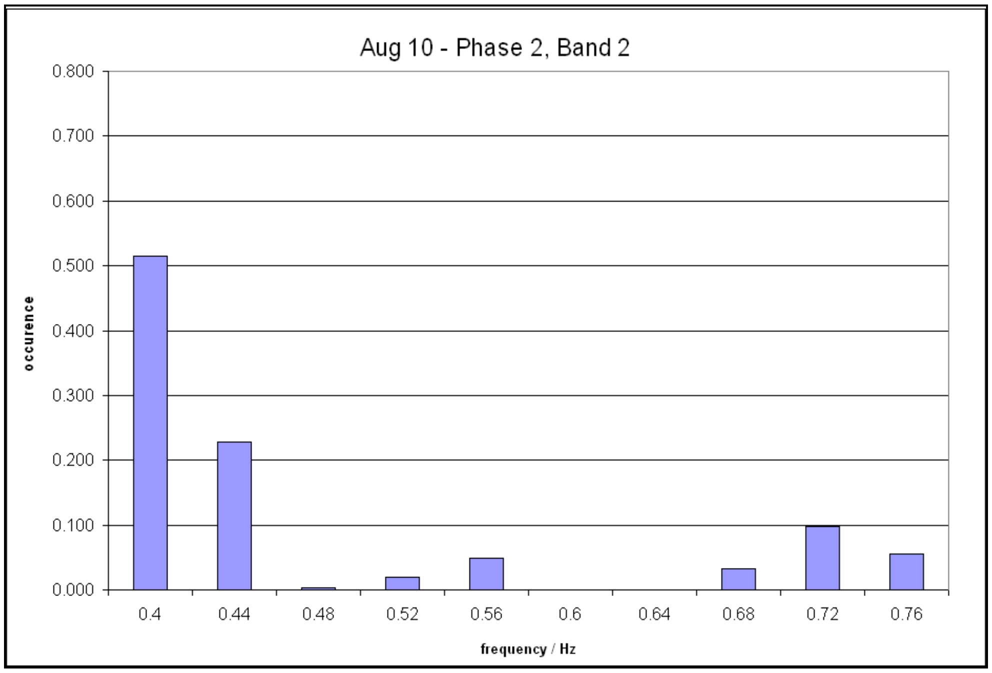

Figure 6-15: Histogram of Maxima, Band 2, Phase 2, August 10, 1996

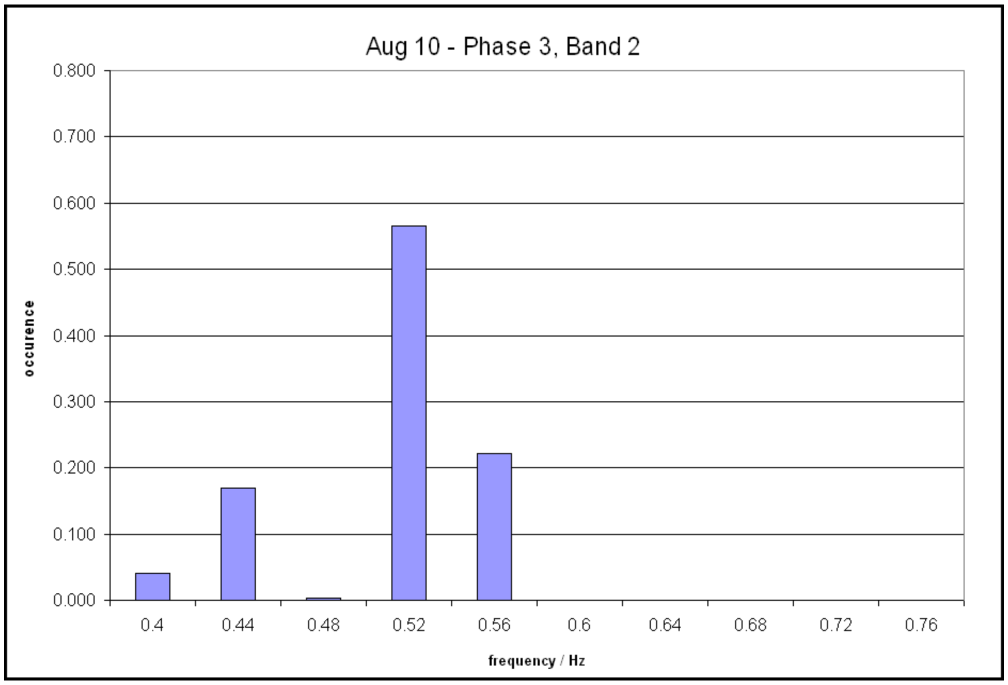

Figure 6-16: Histogram of Maxima, Band 2, Phase 3, August 10, 1996

# 7 Conclusions

The results of the data analysis did not support our hypothesis of detecting impending dynamic instability problems by a set of indicators or features of the grid frequency, which are based on the symptoms of oscillatory behavior in large interconnected power systems. Finding some historic data that are representative of low- and high-stress conditions was difficult. It is almost impossible to determine with any certainty a condition on a large and complex electric power system when the system is under low stress. The randomness and magnitude of constantly changing loads and adjustments by generators to meet the demand, coupled with the randomness of the unplanned outages, which cause changes in the topology of the network, makes it very difficult, if not impossible, to definitively declare a state of the power system as low stress. Even during periods at night, when the load tends to be lower than during the day, it is not obvious that the system attains a low or lower-stress state. Transmission outage, planned or unplanned, may pose a difficult burden on transmission engineers to keep the system in stable and safe condition. Because of the inherent inability to establish a state of low-stress as a reference case, it became difficult during this analysis of historic data to detect the transition from a safe condition to that of an impending problem. Even in the analysis of the data representing the August 10, 1996, breakup, it is not entirely clear what the overall system condition was prior to the Keeler-Allston line break, which appeared to have initiated a sequence of events that led to the system breakup. Much research has been done on the event of August 10, 1996, which provided valuable insights into the systems ability to dampen electro-magnetic oscillations in the system. Researchers were able to determine the damping characteristics of the system after the first event (Keeler-Allston line break). This line break triggered a sufficiently large perturbation to the power flows and excitation to the system, from which the damping coefficient was determined. What is not clear is what the damping characteristics were prior to Keeler-Allston line break. On August 10, 1996, the recorded data clearly indicated that there was a sequence of events that progressively deteriorated the system's ability to recover from the prior system events. Observed was a sliding slope in the "health" of the power system that ultimately resulted in a total system collapse.

A necessary requirement for an effective detection technology is to recognize system conditions as the power system approaches dangerously close the edge of stable and safe operating conditions. Because of the complexity of the power system, the edge of safe operations is a moving target and depends on load conditions and network topology and thus may change from hour to hour. As a result of this data analysis, it appears questionable whether the chosen approach will be successful in the long-run. The major obstacle for this approach is the necessity to establish a reference scenario that would represent safe grid operating conditions. To establish this, a large series of the conditions needs to be analyzed to become familiar with the spectrum of variability for each indicator to establish signatures or patterns for impending problems. The scope of such extended analysis will probably be large and may potentially not be the most effective solution path to the overall objective.

An alternative approach, if feasible, could potentially lead to a promising detection of dynamic instability of the power system. This alternative approach focuses on determining the transfer function that describes the dynamic behavior of the entire power system, from which the standard stability analysis methods can be applied.  So far, no one has successfully established a power system transfer function of sufficient accuracy with which to perform a meaningful stability analysis.  We discuss the major elements and benefits of this approach in the Section 8.

# 8 Recommendations for Future Work

This research provided valuable insights into the complexity and difficulty of identifying impending grid problems. We focused on dynamic instability problems after learning that voltage instability problems may not be identifiable with spectral analysis methods. Several recommendations are offered as part of the lessons learned from this project. They are listed and summarized below and discussed further in Sections 8.1 – 8.3.

4. Under-frequency load control could provide an important grid reliability enhancement. Although reactive in its response, an under-frequency load control strategy with frequency responsive appliances and devices could provide import reserves that are currently furnished by generators that are either already spinning or that can be ramped up in their output. This work and its benefit are discussed further in Section 8.1.

5. The data analysis results did not confirm our hypothesis for detecting grid stress in advance of an event. However, it remains unclear as to whether the primary cause for this result is the approach we used for spectral analysis or the specific data we analyzed. Even for the August 10, 1996, event, it is not clear whether the power system was under high stress prior to the Keeler-Allston line break. To explore the system state further with respect to the stress condition, we recommend an analysis of system conditions several hours, perhaps 1 or 2 days, prior to the Keeler-Allston line break to capture more diversity in the grid condition that may include conditions more characteristic of what we defined as low stress.

6. Enhance fundamental understanding of the stability characteristics of the power system by utilizing system identification techniques that result in a real-time transfer function approximation of the entire power system. If a real-time system transfer function of sufficient accuracy can be established, it would enable the use of standard stability analysis tools for determining distance to the stability edge.

7. For dealing with voltage stability problems – a concern that the CAISO raised – we recommend the use of under-voltage relays for induction motors, as found in compressor motors for air-conditioning systems. We describe some of the underlying mechanism and systems benefits that can be derived from the use of under-voltage relaying.

## 8.1 Under-frequency Load Control using Grid-Friendly Appliances

The electric power grid relies on the rotational (inertial) kinetic energy of the connected synchronous generators to help balance electricity production and consumption. Contained within this inertia is enough energy storage to sustain the grid for cycles to seconds (depending on the amount of imbalance). If there is too much generation, the system frequency increases, too little and the system frequency decreases. Small mismatches between generation and load result in small frequency deviations. These small shifts do not degrade reliability or market efficiency, although large shifts can ultimately lead to system collapse.

Likewise, system frequency provides an indication of the interconnection’s generation/load balance. Frequency can be measured instantly anywhere in the interconnected grid without the need for additional communications. This facilitates dispersed, autonomous response to system casualties by generators and loads. Assuming that all control systems such as automatic generation control (AGC) and speed governors are working correctly, a low system frequency is indicative of a low generation reserve. If frequency deviates from the standard 60 Hz, a range of system reactions takes place as seen in Figure 8-1 .

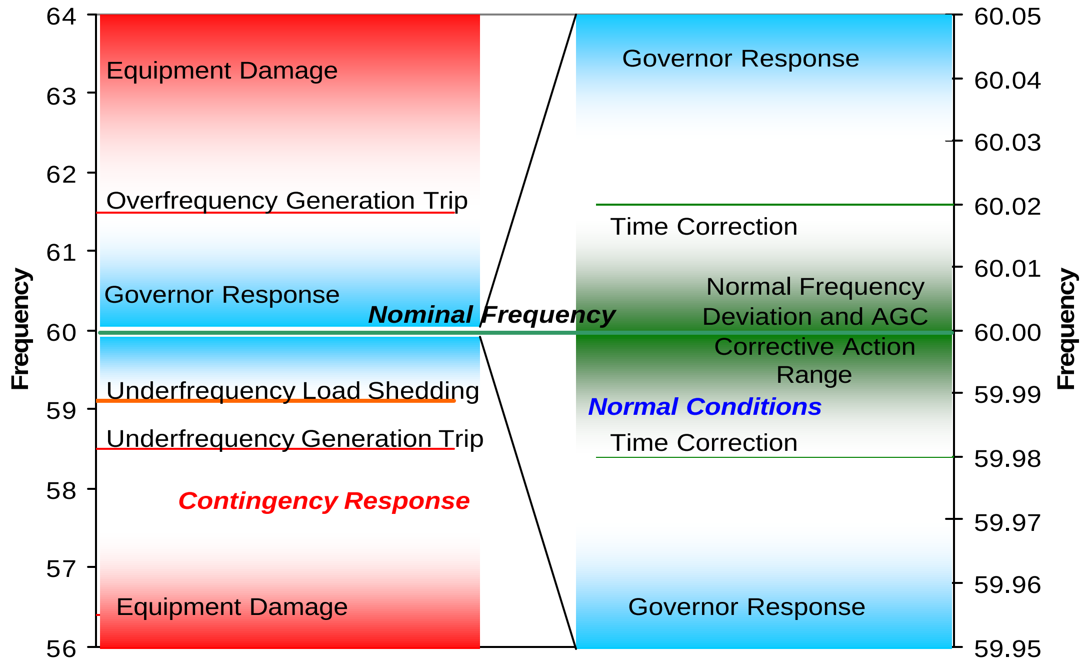

Figure 8-1: Frequency is tightly controlled under normal conditions and coordinated under all conditions.

Notice that far outside the “Normal” frequency range is an under-frequency load-shedding strategy is deployed. This is the first level of system protective response (see Figure 8-1). It is considered a drastic measure because it is invoked at the utility substation level. This type of application accomplishes its objective of maintaining system integrity, but at the cost of cutting off all power to some customers. The load that is dropped at the substation level is not been selected based on significance or convenience. This type of under-frequency load shedding is ‘all or nothing;’ either the feeder is de-energized, dropping all loads on it, or it remains energized, keeping all loads running.

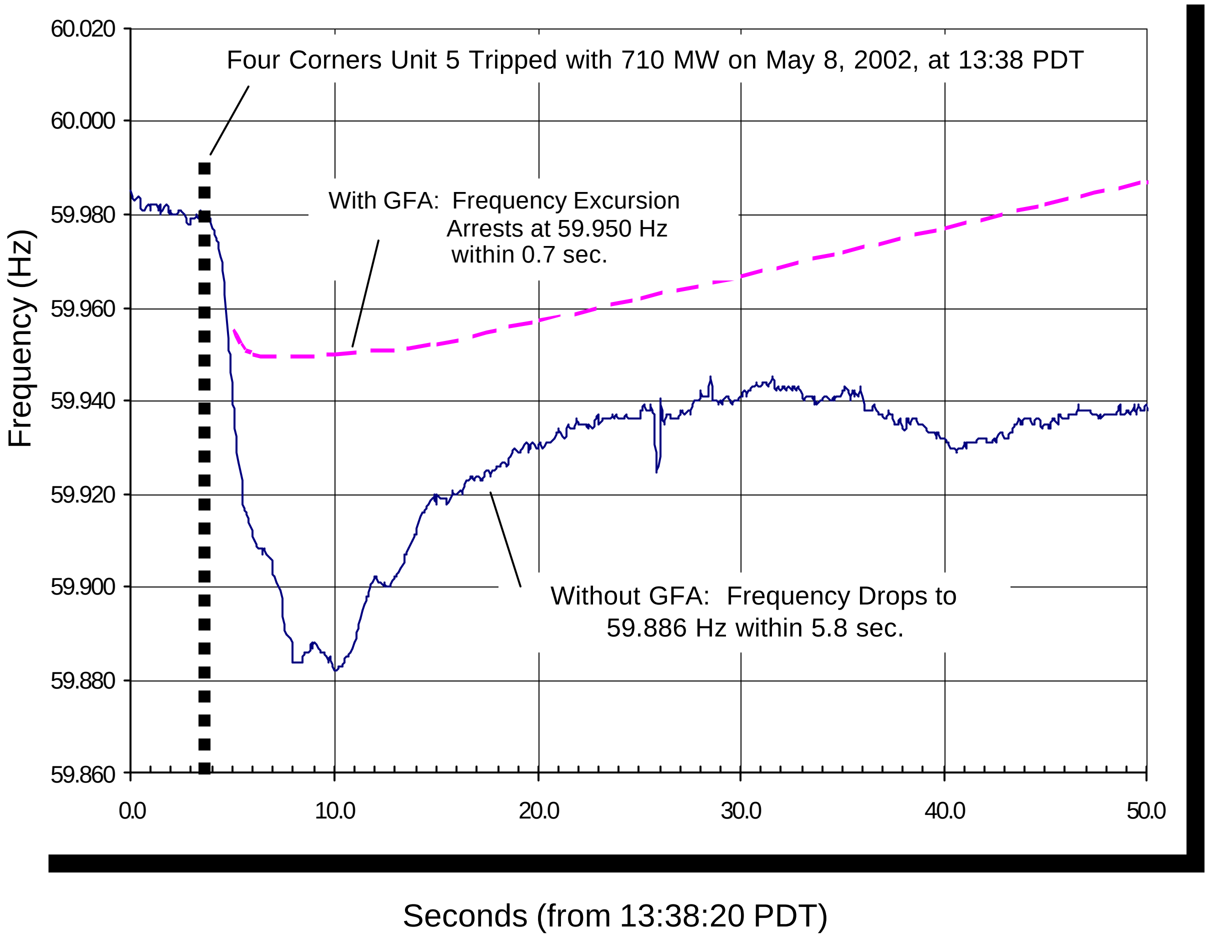

Figure 8-2: Impacts of Frequency-Responsive Loads Using Grid Friendly Appliances

A unique implementation of this idea implements under-frequency load shedding at individual appliances. This type of system load response has already been developed in part by PNNL staff, and has shown exceptional results for minimal economic investment. Instead of the traditional 'all or nothing' approach to load shedding, this method automatically curtails non-essential loads (such as residential appliances) before a crisis develops. Figure 8-2 displays the actual system response to a sudden loss of generation. On the same figure, the simulated

system reaction is plotted for grid-friendly under-frequency protection implemented in large quantities. Notice that the system frequency falls much less than it would have without grid-friendly-appliance (GFA) technology.

Grid-friendly appliances provide a rapid and automatic response to grid crises. Implementation of under-frequency load shedding at the appliance level provides increased power system reliability and security by acting as reserve margin, while going unnoticed by the consumer.

## 8.2 System Identification Approach of the Electric Power System

Techniques used to evaluate the results or ‘symptoms’ of high-stress conditions in the power system have been presented in this paper; however, a deeper analysis can provide information about the condition itself, without relying on the evaluation of its symptoms. Such methods require analysis to convert the noise response of a system, obtained at the wall outlet, into a transfer function, which provides in pole locations for the power system. By measuring the distance of the system poles to the right-hand s-plane (locus of system instability), and by tracking the rate at which those poles move toward the right-hand s-plane, a smart chip can provide a direct indication of actual or probable system instability.

Load switching and other random phenomena in a loosely connected power system produce ambient process noise that contains much useful information about oscillatory dynamics. The extraction of *qualitative* information about system behavior has been a fairly routine matter for many years. The extraction of *quantitative* dynamic information from ambient noise has been less successful. There are a number of reasons for this.

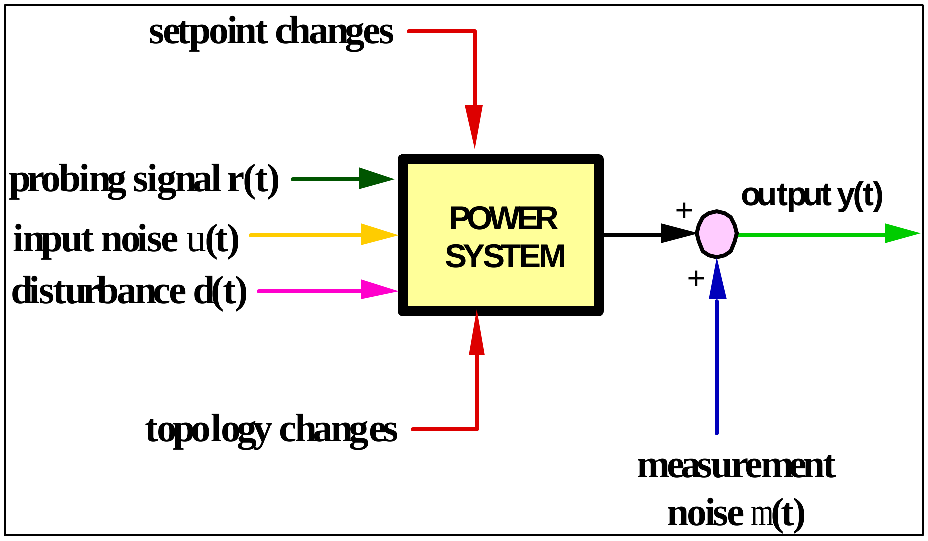

Figure 8-3: Information sources in process identification

Figure 8-3 presents a schematic view of the environment in which such analysis is performed. Input noise v(t) is colored by system dynamics and produces a process noise component in the output y(t). The output also contains a some measurement noise, which may necessitate the use of filtering or better instrumentation. Disturbances, set point changes, and changes in network topology are more serious matters. These often produce important changes in power system dynamics and, even if they do not, they may disrupt the signal analysis process. The object of ambient analysis may be to detect and classify such changes (plus the causal events) when they are not directly observable by other means. Whether or not this is the case, it may still be necessary to detect such changes to properly interpret signal analysis results and perhaps to re-initialize the signal processing. The problem of "hidden inputs" to the system is a harmful one.

The idea in this assessment is that signal processing methods that extract quantitative dynamic information from ambient process noise must, directly or indirectly, derive that information from a numerically estimated autocovariance function [Bendat and Piersol 1993]. To the extent that this conjecture is true, the key issues to examine are the error properties of autocovariance estimates, and the degree to which estimation errors affect the accuracy of estimated oscillatory parameters.

The general finding is that the complexity of power system dynamics can easily "deceive" parametric methods that are based upon autocovariance estimates. Leading problems are undetected exogenous inputs to the system ("hidden inputs"), the large number of closely- spaced oscillatory modes, and the oscillatory nature of estimation errors in forming the autocovariance function. This finding is consistent with results that have been obtained with actual system data over many years. Improved results might be obtainable through the use of multiple signals, through "adaptive" logic that simultaneously adjusts the order of a "parsimonious" model and the fitting window for model construction, and possibly through selective filtering. If such heuristic tuning is carried too far, however, the resulting "ModeMeter" will increasingly reflect built-in tuning assumptions rather than actual system behavior.

## 8.3 Prevention of Stalled Induction Motors

In May 1995, the Sacramento Municipal Utility District (SMUD) issued a report summarizing transmission line outages that resulted in delayed voltage recovery followed by a loss of load. This “local blackout” occurred in August 1994, when four transmission lines in a common corridor went out of service simultaneously, initiating a delayed voltage recovery. The sustained low voltages on the system were caused by the stalling of low-inertia rotating machines (such as air conditioners) during a system disturbance. Conclusions of the preliminary study acknowledged that delayed voltage recovery could indeed occur in the SMUD system. Investigation of measures to prevent motor stalling and delayed voltage recovery was recommended.

When system voltage begins to fall as a result of a line fault or high system loading, the drop in voltage causes induction motors to draw more current to maintain their power output; however, drawing more current perpetuates the drop in voltage. The cycle slowly continues until the voltage level at the terminals of the induction motor can no longer support the operation of the motor, and it stalls. This condition renders the motor essentially useless, with its output power near zero; however, it draws up to six to ten times its rated current, all of which is at an extremely lagging power factor, draining capacitive reactance from the power system. At his point, the system voltage drops sharply and quickly. Other induction motors in near proximity to the voltage collapse, start to stall, and the voltage collapse propagates through the system rapidly.

Large industrial style induction motors typically utilize under-voltage relays for motor protection. Although the primary purpose of these relays is to protect the induction motor itself against the effects of low system voltage, they also protect the power system by tripping the machine off service before a system under-voltage can propagate into a voltage collapse. The protection that these under-voltage relays provide to the power system is a serendipitous benefit, and is not the purpose of the relays.

In the residential setting, there is a major violator with regards to power system voltage instability, the stalling of induction motors associated with home heating, ventilation and air conditioning (HVAC) units. These inexpensive induction motors rarely, if ever, include under-voltage or stalled-motor protection. The lack of such protection is justified based on the risk of losing the operability of the induction motor versus the cost of integrating the protective device. In fact, as a rule, protection devices are designed and installed only to protect the equipment they serve. Because these small residential induction motors pose little financial consequence if they fail, they are fitted with thermal over-current protection only. This inexpensive protective device relies on over-temperature sensors to identify a faulty or malfunctioning condition and responds by disconnecting the motor from the electric power system. Thermal protection serves the motor adequately; however, it does too little, too late to benefit the power system during a voltage collapse. Because the thermal devices take 10 seconds or more before they trip a stalling induction motor off line, these devices do not sufficiently mitigate power system voltage collapse or a delayed voltage recovery.

Several inexpensive methods of stall detection can be implemented into residential HVAC induction motor units. One such method utilizes phase delay between the source and winding of the induction motor. This phase delay occurs during normal operation but does not occur during stalled conditions. Such a detector could be easily integrated into inexpensive induction motors, and in large quantities, would have profound impact by mitigating delayed voltage recovery and loss of loads as a result of these effects.

# 9 References

Advantech. 2003. Datasheet on the 486 Mini Biscuit Single Board Computer CPC-2245N. Advantech Company. Irvine, California. Available on the Internet at http://partner.advantech.com.tw/epartner/Files/Temp/1-1OR4R-2.pdf.

Altera. 2003. Datasheet on MAX 700B Programmable Logic Device. Altera Corporation. San Jose, California. Available on the Internet at http://www.altera.com/literature/ds/m7000b.pdf.

ANSI. 1988. Draft Proposed American National Standard for Information Systems -- Programming Language C. Technical Report X3J11/88-158, Accredited Standards Committee, X3 Information Processing Systems, American National Standards Institute, December.

Bendat, J. and A. Piersol. 1993. Engineering Applications of Correlation and Spectral Analysis. John Wiley & Sons, New York.

Borland. 1989. Borland Turbo C. Version 2.01. Borland Software Corporation. Scotts Valley, California. May 1989.

CAISO. 1998. 1998 California Operating Studies Subcommittee (OSS) Handbook. Prepared by Chuck-yan Wu. Revision 2.0.0. California Independent System Operator, Folsom, CA. March 1998.

CAISO. 2002. AC/DC Nomogram for North-to-South Flow and COI Nomogram for South-to-North flows. Procedure No. T-116. Version 4.5. Effective Date June 5, 2002. California Independent System Operator.

CEC. 1997. "Survey of the Implications to California of the August 10, 1996, Western States Power Outage." Report of the California Energy Commission, June 1997. Available on the Internet at http://www.energy.ca.gov/electricity/index.html#reliability.

GE. 2001. GE Power Systems Energy Consulting. Commands Reference Manual, Version 12.0. GE Power Systems Energy Consulting, Schenectady, NY, March, 2001.

Groll, J., U. Grüner, and H. Wiese. 1998. C als erste Programmiersprache (ISO-Standard), B. G. Teubner Verlag, Stuttgart, Germany.

HAMEG. 2003. Datasheet on Universal Counter 8021-3. HAMEG GmbH. Frankfurt, Germany. Available on the Internet at http://www.hameg.de/de/index.htm.

Hauer. 2001. Enhanced Information Resources for Managing Reliability and Performance of the Western Power System. Presentation by John Hauer, Pacific Northwest National Laboratory, Fall 2001.

Hauer, J. and J. Dagle. 1999. Review of Recent Reliability Issues and System Events. PNNL Technical report PNNL-13150, prepared for the U.S. Department of Energy Transmission Reliability Program by the Consortium for Electric Reliability Solutions (CERTS). Pacific Northwest National Laboratory, Richland, Washington.

Hirst, E. and B. Kirby. 2003. *Technical Issues Related to Retail-Load Provision of Ancillary Services*, New England Demand Response Initiative, February 2003. Available at http://www.ehirst.com/PDF/NEDRIReservesBackground.pdf.

HP. 2003. Datasheet on 53131A Universal Frequency Counter. Agilent Technologies (formerly Hewlett-Packard Company), Palo Alto, California. Available on the Internet at http://cp.literature.agilent.com/litweb/pdf/5967-6039EN.pdf.

Kirby, B. 2003. Spinning Reserve From Responsive Loads. ORNL/TM-2003/19. Oak Ridge National Laboratory, Oak Ridge, Tennessee.

Kirby, B. and M. Alley. 2002. Spinning Reserves from Controllable Packaged Through the Wall Air Conditioner (PTAC) Units. ORNL/TM-2002/xx. Oak Ridge National Laboratory, Oak Ridge, Tennessee.

NERC. 2002. Policy 1 – Generation Control Performance, Section C. Frequency Response and Bias NERC Guides, North American Electric Reliability Council, Princeton, New Jersey.

Press, P.W. B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. 1993. *Numerical Recipes in C.* Cambridge University Press, Cambridge.

VanZandt, V. R., M.J. Landauer, W.A. Mittelstadt and D.S. Watkins. 1997. "A Prospective Look at Reliability with Lessons From the August 10, 1996, Western System Disturbance," Proceedings of the International Electric Research Exchange Workshop on Future Directions in Power System Reliability, Palo Alto, CA, May 1-2, 1997.

WSCC. 1996. "Western Systems Coordinating Council Disturbance Report for the Power System Outage that Occurred on the Western Interconnection August 10, 1996, 1548 PAST." Approved by the WSCC Operations Committee on October 18, 1996. Available on the Internet at http://www.wscc.com/outages.htm.

# Appendix A: Description of Frequency Sensor

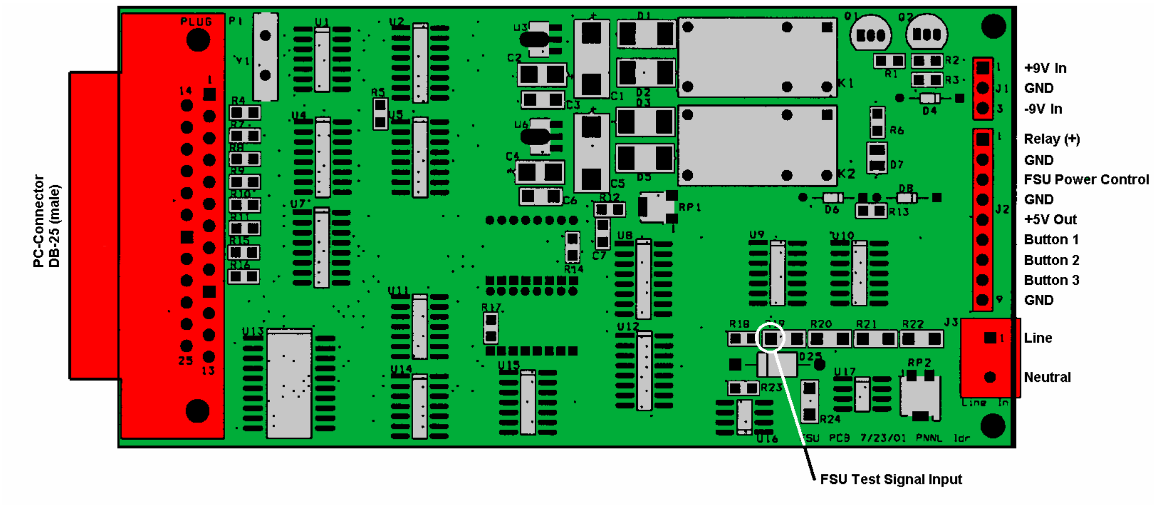

Figure A-1: Printed Circuit Board of Frequency Sensor

Table A-1: Component List of Frequency Sensor Hardware

| Item | Qty | Reference | Part |
|---|---|---|---|
| 1 | 2 | C1, C5 | 47µ / 16V |
| 2 | 2 | C2, C4 | 4.7µF |
| 3 | 19 | C3, C6, C8, C9, C12, C13, C15, C16, C17, C18, C19, C20, C21, C22, C23, C24, C25, C26, C27 | 100nF |
| 4 | 2 | C28, C7 | 47pF |
| 5 | 2 | C10, C11 | 33pF |
| 6 | 1 | C14 | 10pF |
| 7 | 4 | D1, D2, D3, D5 | 1N4004 |
| 8 | 3 | D4, D6, D8 | 1N4148 |
| 9 | 1 | D7 | GREEN LED |
| 10 | 16 | D9, D10, D11, D12, D13,D14, D15, D16, D17, D18, D19, D20, D21, D22, D23, D24 | HS1001 |
| 11 | 1 | D25 | SA6.0CA |
| 12 | 1 | J1 | .1 x 3 Single Row Header |
| 13 | 1 | J2 | .1 x 9 Single Row Header |
| 14 | 1 | J3 | 2 pin screw terminal |
| 15 | 2 | K2, K1 | Aromat DR-6V |
| 16 | 1 | P1 | CONNECTOR DB25 |

| Item | Qty | Reference | Part |
|---|---|---|---|
| 17 | 1 | Q1 | 2N3904 |
| 18 | 1 | Q2 | 2N3906 |
| 19 | 1 | RP1 | 10K |
| 20 | 1 | RP2 | 100K |
| 21 | 1 | R1 | 180K |
| 22 | 1 | R2 | 56K |
| 23 | 3 | R3, R5, R26 | 4.7K |
| 24 | 8 | R4, R7, R8, R9, R10, R11, R15, R16 | 470 |
| 25 | 2 | R25, R6 | 2.2K |
| 26 | 1 | R12 | 6.8K |
| 27 | 1 | R13 | 3.3K |
| 28 | 2 | R17, R14 | 1K |
| 29 | 1 | R18 | 1.2K |
| 30 | 4 | R19, R20, R21, R22 | 11.3K |
| 31 | 2 | R23, R28 | 10K |
| 32 | 1 | R24 | 4.7M |
| 33 | 1 | R27 | 100K |
| 34 | 2 | U14, U1 | CD74AC04 |
| 35 | 4 | U2, U4, U5, U7 | CD74AC161 |
| 36 | 1 | U3 | 78L05 SOT89 |
| 37 | 1 | U6 | 79L05 SOT89 |
| 38 | 1 | U8 | SN74HC4538 |
| 39 | 1 | U9 | SN74HC04 |
| 40 | 1 | U10 | SN74HC00 |
| 41 | 1 | U11 | SN74HC11 |
| 42 | 1 | U12 | CD74AC109 |
| 43 | 1 | U13 | CD74HC573 |
| 44 | 1 | U15 | SN74HC32 |
| 45 | 1 | U16 | LT1394 |
| 46 | 1 | U17 | LM6261 |
| 47 | 1 | Y1 | 4.9152 MHz |

# Appendix B: Source Code for Controllers and Analysis Platform Software

## Code for Controls Software

This appendix contains the source code for the first and second generation controllers.

## First Generation Controls Software

The controls software consists of the following modules in Borland Turbo C language[6]:

- FSU.C
- FSU-DEF.C
- FSU-LCD.C
- FSU-VAR.C
- FSU-SUB.C

Each module is listed below. The function of each module is described in its header.

```
/****************************************************************************/
/*                                                                          */
/* File "fsu.c", created on 04/28/2001 by Daniel L. Oedingen, PNNL         */
/*                                                                          */
/* ----------------------------------------------------------------------- */
/*                                                                          */
/* Current program version  : See "#define PROG_VERSION" in "fsu-def.c".    */
/*                                                                          */
/* Last updated             : 07/27/2001 by DLO                             */
/*                                                                          */
/* Compile with             : "tcc -B -G -1 -a -f87 fsu.c" (Borland Turbo C */
/*                            Compiler Version 2.01 or higher); see also    */
/*                            further documentation for compile options for */
/*                            configuring this software for data logger     */
/*                            modes etc.                                    */
/*                                                                          */
/* The TCC options mean     : -B..... Compile via assembly (TASM)           */
/*                            -G..... Generate for optimized speed          */
/*                            -1..... Use 8086/80186 instruction set        */
/*                            -a..... Generate word aligned object code     */
/*                            -f87... enable 8087 floating point support    */
/*                                                                          */
/* Supported platforms      : - MS-DOS 5.0 or higher, Win 3.x               */
/*                            - Microsoft Windows 95/98/ME (all OSRs)       */
/*                                                                          */
/*                            Please note that the NT-based platforms of    */
/*                            the Windows operating system (Windows NT,     */
/*                            Windows 2000) are NOT supported, because they */
/*                            block direct hardware access which is needed  */
/*                            to control the FSU / printer port hardware.   */
```

[6] Borland, 1989. Borland Turbo C. Version 2.01. Borland Software Corporation, Scotts Valley, California, May 1989.

```
/*                                                                    */
/* Functionality            : This program performs test data readings from */
/*                            the Frequency Sensing Unit developed for the  */
/*                            "Grid-Friendly Appliances" via an ECP/EPP-    */
/*                            compatible parallel port of an ordinary PC.   */
/*                            Special file versions for data logging are    */
/*                            also available (just enable the 'DATA_LOGGER' */
/*                            label in "fsu-def.c" and re-compile "fsu.c"). */
/*                            If you want to activate the event-driven data */
/*                            logger mode, enable the 'EVENT_DRIVEN' label. */
/*                            Support for a serial SEETRON LC-Display and   */
/*                            three push-buttons is also available in order */
/*                            to operate the GFA independently from a CRT,  */
/*                            a keyboard etc.                               */
/*                                                                    */
/* Features implemented yet : - ECP/EPP port compliance test routine        */
/*                            - Routines for installation and removal of    */
/*                              the interrupt service handler on IRQ7       */
/*                            - Power relay control (/INIT-line)            */
/*                            - Automatic- and manual mode controls (over-  */
/*                              rides automatic load control)               */
/*                            - Main loop syncronization with Interrupt     */
/*                              Service Routine                             */
/*                            - Washout filter for spike elimination        */
/*                            - Slope detection to sense frequency bounces  */
/*                              within the band defined by the software     */
/*                              thresholds                                  */
/*                            - Fixed- / randomized switching delay to turn */
/*                              off the load for a certain time after a     */
/*                              slope has been detected.                    */
/*                            - A rotating memory to decrease the data      */
/*                              logger output (event-driven mode). This     */
/*                              mechanism can also be used as the data      */
/*                              input for a Fast Fourier Transform.         */
/*                            - OPERATION MODE CONTROL: Adjust the 'Compi-  */
/*                              ler Settings' section in "fsu-def.c" to     */
/*                              configure the software for one of the       */
/*                              following modes of operation:               */
/*                                - 'Normal' GFA Control Logic Mode (g.bat) */
/*                                - Event-Driven Data Logger Mode (e.bat)   */
/*                                - Continuous Data Logger Mode (c.bat).    */
/*                            - Built-in LCD screen output functionality.   */
/*                              Simply define the "LCD" label in file       */
/*                              "FSU-DEF.C" to enable LCD messages (the     */
/*                              stdout device, i.e. Video Card / CRT will   */
/*                              display no messages in this mode except     */
/*                              from a notice that the LCD is enabled).     */
/*                            - FSU hardware power is now turned on when    */
/*                              the software is started and turned off on   */
/*                              program exit (necessary to avoid problems   */
/*                              occuring while the program is started,      */
/*                              especially in a Windows 9x DOS-Box which    */
/*                              tends not to work properly in some cases if */
/*                              the FSU hardware power is already on when   */
/*                              the software is loaded).                    */
/*                            - Added 3 push-buttons to control the GFA in  */
/*                              LCD mode when no PC keyboard is available.  */
/*                              Button 1: Toggles between automatic- and    */
/*                                        manual mode (substitutes "a"- and */
/*                                        "m" keys).                        */
/*                              Button 2: Displays the software settings    */
/*                                        menu (no actions will be taken    */
/*                                        during this time); press the same */
/*                                        button again to return to pre-    */
/*                                        vious screen.                     */
/*                              Button 3: Exits the program or confirms     */
/*                                        messages (e.g. if EPP port com-   */
/*                                        pliance test is activated).       */
/*                                                                          */
/* Special Program versions : - For grid frequency sensing and graphical    */
/*                              spectral output use the file "fsuspect.c".  */
/*                              This version is completely independent from */
/*                              any other FSU source code (no *.c-includes  */
/*                              and stuff) and is understood as a not offi- */
/*                              cially supported version which has just     */
/*                              been created for test purposes (provided    */
/*                              "as is"). Because this software uses the    */
/*                              BGI (Borland Graphics Interface), you will  */
/*                              have to load the source file into the TC    */
/*                              editor (e.g. by typing 'tc fsuspect' at the */
/*                              DOS prompt) and compile it by pressing F9.  */
/*                              Be also sure to have a copy of the Borland  */
/*                              graphics driver ("EGAVGA.BGI") in the same  */
/*                              directory as the executable file.           */
/*                                                                          */
/* Related files            : - See file "fsu-sub.c" for implementation     */
/*                              of the subroutines used in this software.   */
/*                            - Any symbolic constant, including user       */
/*                              program settings, can be found in file      */
/*                              "fsu-def.c" (except from all this LCD-      */
/*                              related stuff).                             */
/*                            - The declaration of the global variables is  */
/*                              located in file "fsu-var.c".                */
/*                            - Any LCD-related code, symbolic constant and */
/*                              variable is located in file "fsu-lcd.c".    */
/*                                                                          */
/****************************************************************************/

/*-- Definition of symbolic constants ---------------------------------------*/

#include "fsu-def.c"

/*-- Include C language header files ----------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <dos.h>

/*-- Declaration of global variables ----------------------------------------*/

#include "fsu-var.c"

/*-- Include source code file for subroutines / LCD-support ----------------*/

#ifdef LCD
  #include "fsu-lcd.c"
#endif

#include "fsu-sub.c"

/****************************************************************************/

/*-- Main program -----------------------------------------------------------*/
```

```
int main (void)
{
  #ifdef DEBUG
    #ifndef LCD
      system ("cls");
      HideCursor ();

    #else
      printf  ("\n   Generating  screen  output  for  SEETRON  LC-Display  (type  G12864
V2.0)...");
      printf ("\n\n  Press <ESC> (i.e. push-button #3) to abort program.");
      Init_LCD_COM1 ();
      LCD_Small_Font_Mode ();
    #endif
  #endif

  #ifdef DATA_LOGGER
    #ifndef EVENT_DRIVEN
      /* Save time stamp of the moment the program has been started         */
      startup_tstamp = time (NULL);
    #endif
  #endif

  buf_ptr = buffer;       /* Set pointer to 0. element of buffer [BUF_SIZE] */
  srand (NULL);           /* Initialize C random number generator           */

  /* Enable GFA control logic on program startup (i.e. the load is turned   */
  /* off and user override mode is disabled). The data logger versions do   */
  /* not provide any relay control functions beyond this point.             */

  outp (LPT1_CONTROL_PORT, inp (LPT1_CONTROL_PORT) & LOAD_OFF);

  /*-- Display program information -----------------------------------------*/

  #ifndef SKIPINTRO
    DisplayStartupInfo ();
  #else
    fprintf (stderr, "\n");
  #endif

  /*-- Test if parallel port is ECP/EPP compliant --------------------------*/

  #ifndef SKIPTEST
    TestPortCompliance (); /* If SKIPTEST is not defined, the ECP/EPP       */
                          /* hardware compliance test is performed.       */
  #else

    /* Otherwise just configure port for data input */

    outp (LPT1_CONTROL_PORT, inp (LPT1_CONTROL_PORT) | ENABLE_TRISTATE_MODE);
  #endif

  /*-- If test was successful, install Interrupt Service Routine (ISR) -----*/

  Install_ISR ();

  /* Turn FSU hardware power on */

  delay (500);
  outp (LPT1_CONTROL_PORT, inp (LPT1_CONTROL_PORT) & FSU_ON);

  /*-- Start reading data from LPT1 ----------------------------------------*/
  #ifdef DEBUG
    #ifndef LCD
      printf ("\n\n  --------------------------------------------------------------
-------------");
      printf ("\n  USER INFO       : Please connect FSU to your computer's parallel
port now.");
      printf ("\n\n  AUTOMATIC MODE : Valid keys are <a>uto mode, <m>anual mode or
<ESC> to quit.");
      printf ("\n\n  RESULT          : <Waiting for valid input frequency (%5.3f ...
%5.3f  Hz)>.",  LOWER_FREQ_THRESHOLD  +  FREQ_HYSTERESIS,  UPPER_FREQ_THRESHOLD  -
FREQ_HYSTERESIS);
      printf ("\n  ----------------------------------------------------------------
-----------");
      delay (1000);
    #else
      Init_GFA_Screen ();
    #endif
  #endif

  /*-- Data reading / processing section -----------------------------------*/

  do
  {
    if (data_avail != 0)
    {
      #ifdef PROCESS_CHANGES_ONLY
       if (data != old_data)
       {
      #endif

       #ifdef FSU_DETECTION
         if (data == 255)         /* Every data bit = 1 for a longer time  */
                              /* means that the FSU is most likely not */
                              /* connected at the moment              */
         {
           gotoxy (RESULT_OUTPUT_X, RESULT_OUTPUT_Y);
           printf ("<ERROR : FSU not detected>.                               ");
         }
         else
         {
       #endif

         if (data < 128)        /* Bit 7=0: Hardware range miss detected   */
         {
             range_miss++;
             if (range_miss >= MAX_RANGE_MISS_NO)  /* How many in a row?     */
             {
              load_state = load_state & 0xFD;     /* Prepare load state OFF */

             #ifndef DATA_LOGGER
               #ifdef DEBUG
                #ifndef LCD
                  gotoxy (RESULT_OUTPUT_X, RESULT_OUTPUT_Y);
                  printf      ("<ERROR      :      Range      miss      detected>.
");
                #else
                  if (message_state != 1)
                  {
                    LCD_Printf ("??.???", 11, 5);
                    LCD_Printf ("RANGE MISS detected.                    ", 1, 7);
                    message_state = 1;
                  }
                #endif
               #endif
             #else
               #ifdef EVENT_DRIVEN
                gettime (&t);
                fprintf  (stderr,  "     <ERROR  :  Range  miss  detected>  at
 %2.2d:%2.2d:%2.2d,%2.2d ", t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
                getdate (&d);
                fprintf  (stderr,  "on  %2.2d/%2.2d/%4.4d  %s  %2.1d\n",  d.da_mon,
 d.da_day, d.da_year, LOGGER_MODE, event_count);
                printf ("<ERROR : Range miss detected>.\n");
               #else
                gettime (&t);
                fprintf  (stderr,  "     <ERROR  :  Range  miss  detected>  at
 %2.2d:%2.2d:%2.2d,%2.2d ", t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
                getdate (&d);
                fprintf  (stderr,  "on  %2.2d/%2.2d/%4.4d  %s\n",  d.da_mon,  d.da_day,
 d.da_year, LOGGER_MODE);
                printf ("<ERROR : Range miss detected>.\n");
               #endif
              #endif
            }
         }
         else
         {
           range_miss = 0;   /* Any valid frequency value resets the range */
                             /* miss counter; this prevents malfunctions   */
                             /* on just a few range miss messages reported */
                             /* by the hardware due to crosstalk, grid     */
                             /* noise and other disturbances, especially   */
                             /* on the crappy hand-wired FSU prototype.    */

           if ((data & 0x40) == 0) /* Bit 6=0? Consequence: f <= 60.000 Hz */
           {
             /* Now it gets tricky: Bit 7 of 'data' contains no infor-     */
             /* mation about the frequency value read, thus it is set to   */
             /* zero by the '& 0x7F' command. Because D0-D6 of 'data'      */
             /* represent Q1-Q7 of the FSU counter (Q0 is skipped), 'data' */
             /* has to be multiplied by 2 (corresponds a binary shift-     */
             /* left-operation). The 'OFFSET' is added in order to recon-  */
             /* struct the whole divisor stored in the counter during the  */
             /* current read-out process.                                  */

             divisor = OFFSET + (data & 0x7F) * 2;
           }
           else                    /* Bit 6 = 1 means f is > 60.000 Hz    */
           {
             /* Same procedure, but first convert 2-complement to unsigned */
             /* char- and separate sign representation. Setting Bit 7 to   */
             /* zero is implied in this conversion.                        */

             divisor = OFFSET - ((unsigned char) (~data + 1)) * 2;
           }

           frequency = (double) REF_FREQUENCY / divisor;

           /* Calculate average value if this feature is activated (i.e.   */
           /* 'AVG_VALUE_NO' is > 1) and skip the following code; other-   */
           /* wise proceed with storing the new value in the buffer array. */

           if (avg_count < AVG_VALUE_NO)
           {
             avg_count++;
             freq_sum += frequency;
             if (avg_count >= AVG_VALUE_NO) goto result_output;
           }
           else
           {
           result_output:
             frequency = freq_sum / AVG_VALUE_NO;

             freq_sum = 0.0;            /* Reset variables for next average */
             avg_count = 0;             /* calculation loop                 */

             /* Store frequency value and timestamp in 'buffer [BUF_SIZE]' */

             if (buf_ptr == buffer + BUF_SIZE) buf_ptr = buffer;

             sample_time = time (NULL);
             gettime (&t);

             (*buf_ptr).dataset_frequency = frequency;
             (*buf_ptr).dataset_timestamp_sec = sample_time;
             (*buf_ptr).dataset_timestamp_hund = t.ti_hund;
             buf_ptr++;

             /* Display the result and a message which action is taken     */

             #ifndef DATA_LOGGER
              #ifdef DEBUG
               if (time (NULL) >= slope_time + timeout)
               {
                 #ifndef LCD
                   gotoxy (RESULT_OUTPUT_X, RESULT_OUTPUT_Y);
                   printf     ("Reading     frequency     value     %6.4f     Hz.
 ", frequency);
                 #else
                   sprintf (frequency_dummy, "%6.3f", frequency);
                      LCD_Printf (frequency_dummy, 11, 5);

                      if ((frequency >= LOWER_FREQ_THRESHOLD + FREQ_HYSTERESIS) &&\
                          (frequency <= UPPER_FREQ_THRESHOLD - FREQ_HYSTERESIS) &&\
                          (load_state != 0))
                      {
                        if (message_state != 2)
                        {
                          LCD_Printf ("Nothing to report...                    ", 1,
 7);
                          message_state = 2;
                        }
                      }
                 #endif
               }
               else
               {
                 #ifndef LCD
                   gotoxy (RESULT_OUTPUT_X, RESULT_OUTPUT_Y);
                    printf ("<Slope detected : Turning load off for %d seconds>.
 ", timeout);
                 #else
                   sprintf (frequency_dummy, "%6.3f", frequency);
                      LCD_Printf (frequency_dummy, 11, 5);
                   if (message_state != 3)
                   {
                     LCD_Printf ("SLOPE: Turning load off for ", 1, 7);
                     sprintf (timeout_dummy, "%3.3d", timeout);
                     LCD_Printf (timeout_dummy, 9, 8);
```

```
                      LCD_Printf (" seconds.", 12, 8);
                      message_state = 3;
                    }
                    else
                    {
                      if (slope_time != slope_old)
                      {
                        sprintf (timeout_dummy, "%3.3d", timeout);
                        LCD_Printf (timeout_dummy, 9, 8);
                        slope_old = slope_time;
                      }
                    }
                  #endif
                }
              #endif
             #else
               #ifdef EVENT_DRIVEN
                /* The following will be displayed on the screen just for */
                /* monitoring the data logging process                     */

                fprintf   (stderr,   "      f=%6.4f   Hz   [%10.10ld%2.2d]         at
%2.2d:%2.2d:%2.2d,%2.2d ", frequency, sample_time, t.ti_hund, t.ti_hour, t.ti_min,
t.ti_sec, t.ti_hund);
                getdate (&d);
                fprintf  (stderr,  "on  %2.2d/%2.2d/%4.4d  %s  %2.1d\n",  d.da_mon,
d.da_day, d.da_year, LOGGER_MODE, event_count);
              #else
                fprintf   (stderr,   "      f=%6.4f   Hz   [%10.10ld%2.2d]         at
%2.2d:%2.2d:%2.2d,%2.2d ", frequency, sample_time, t.ti_hund, t.ti_hour, t.ti_min,
t.ti_sec, t.ti_hund);
                getdate (&d);
                fprintf (stderr, "on %2.2d/%2.2d/%4.4d %s\n", d.da_mon, d.da_day,
d.da_year, LOGGER_MODE);

                /* This generates the frequency- and time stamp log file  */
                /* output mentioned above if you use the MS-DOS command   */
                /* for redirecting the stdout-stream into a log file by   */
                /* typing e.g."fsu > data000.fsu" at the MS-DOS-Prompt.   */

                printf ("%6.0f %10.10ld%2.2d\n", frequency * 10000, sample_time,
t.ti_hund);
              #endif
             #endif

             #ifndef DATA_LOGGER

              /* Just a little washout filter to suppress smaller jumps   */
              /* in the grid frequency signal                             */

              if (fabs (frequency - old_freq0) < (double) WASHOUT_FILTER_THRES)
              {
                /* Check if frequency is within user-defined boundaries   */
                /* (see specification of software thresholds in file      */
                /* "fsu-def.c").                                          */

                if   ((frequency   <   LOWER_FREQ_THRESHOLD)   ||   (frequency   >
UPPER_FREQ_THRESHOLD))
                {
                  load_state = load_state & 0xFD;       /* Load state OFF */

                  #ifdef LCD
                    if (frequency < LOWER_FREQ_THRESHOLD)
                    {
```

```
                    if  ((message_state  !=  4)  &&  (time  (NULL)  >=  slope_time  +
timeout))
                    {
                      LCD_Printf ("Under-frequency  loadshedding  active...   ",  1,
7);
                      message_state = 4;
                    }
                  }
                  else
                  {
                    if  ((message_state  !=  5)  &&  (time  (NULL)  >=  slope_time  +
timeout))
                    {
                      LCD_Printf ("Over-frequency  load  shedding  active...   ",  1,
7);
                      message_state = 5;
                    }
                  }
                #endif
              }
              else
              {
                if ((frequency > LOWER_FREQ_THRESHOLD + FREQ_HYSTERESIS) &&\
                   (frequency < UPPER_FREQ_THRESHOLD - FREQ_HYSTERESIS))
                {
                  load_state = load_state | 0x02;     /* Load state ON  */
                }
              #ifdef LCD
                else
                {
                  if (frequency < 60.0)
                  {
                    if  ((message_state  !=  6)  &&  (time  (NULL)  >=  slope_time  +
timeout))
                    {
                      LCD_Printf ("Under-frequency      hysteresis  range...  ",  1,
7);
                      message_state = 6;
                    }
                  }
                  else
                  {
                    if  ((message_state  !=  7)  &&  (time  (NULL)  >=  slope_time  +
timeout))
                    {
                      LCD_Printf ("Over-frequency       hysteresis  range...  ",  1,
7);
                      message_state = 7;
                    }
                  }
                }
              #endif
              }

            /* Wait until 3 values in a row are in the nominal range.   */
            /* This is only necessary for normal mode (to prevent the   */
            /* logic from detecting a slope during program startup      */
            /* which would cause the timeout to trigger immediately).   */

            if ((startup == 1) &&\
                (frequency > LOWER_FREQ_THRESHOLD + FREQ_HYSTERESIS) &&\
                (frequency < UPPER_FREQ_THRESHOLD - FREQ_HYSTERESIS) &&\
                (old_freq0 > LOWER_FREQ_THRESHOLD + FREQ_HYSTERESIS) &&\
                (old_freq0 < UPPER_FREQ_THRESHOLD - FREQ_HYSTERESIS) &&\
                (old_freq1 > LOWER_FREQ_THRESHOLD + FREQ_HYSTERESIS) &&\
                (old_freq1 < UPPER_FREQ_THRESHOLD - FREQ_HYSTERESIS)) startup = 0;
           #else
            startup = 0;
           #endif

           #ifndef DATA_LOGGER
            }
           #endif

           /* Detection of slopes in the frequency signal which are      */
           /* larger in size (e.g. in case of a tripping generator or    */
           /* transmission line and all this stuff), but still within    */
           /* the band between the software thresholds.                  */

           if ((startup == 0) && ((frequency > LOWER_FREQ_THRESHOLD) &&\
                (frequency < UPPER_FREQ_THRESHOLD)) &&\
              ((old_freq0     >     LOWER_FREQ_THRESHOLD)     &&     (old_freq0     <
           UPPER_FREQ_THRESHOLD)) &&\
              ((old_freq1     >     LOWER_FREQ_THRESHOLD)     &&     (old_freq1     <
           UPPER_FREQ_THRESHOLD)) &&\
              (fabs (old_freq1 - frequency) >= SLOPE_DETECTION_THRES) &&\
              (((old_freq0 > frequency) && (old_freq0 < old_freq1)) ||\
              ((old_freq0 < frequency) && (old_freq0 > old_freq1))))
           {
            load_state = load_state & 0xFD;           /* Load state OFF */
            after_slope = 1;         /* Turn on "after slope"-machinery */

            /* Save the present time (when timeout begins) and generate */
            /* a variable timeout                                       */

            #ifndef DATA_LOGGER
              slope_time = time (NULL);
              timeout = FIXED_TIMEOUT + rand () % MAX_RANDOM_TIMEOUT;
            #endif
           }

           #ifndef DATA_LOGGER

            /* Keep load off during timeout after a slope occured       */

            if (!(time (NULL) >= slope_time + timeout) && (startup == 0))
            {
              load_state = load_state & 0xFD; /* Prepare load state OFF */
            }
           #endif

           /* Read the other half of the buffer's size after an event    */
           /* occured, otherwise do nothing. Purpose: Creating a record  */
           /* which contains the event (in the middle) and half of the   */
           /* buffered data on both sides).                              */

           if (after_slope > 0)
           {
            if (after_slope <= (unsigned int) (BUF_SIZE / 2 - 1)) after_slope++;
            else
            {
              #ifdef EVENT_DRIVEN
                event_count++;         /* Increase event counter by one */

                /* Create log file output (writes a record only after   */
                /* an event has been recognized and processed           */
```

```
                  for (after_slope = 0; after_slope < BUF_SIZE; after_slope++)
                  {
                    if (buf_ptr == buffer + BUF_SIZE) buf_ptr = buffer;
                     printf ("%6.0f %10.10ld%2.2d\n", (*buf_ptr).dataset_frequency *
10000, (*buf_ptr).dataset_timestamp_sec, (*buf_ptr).dataset_timestamp_hund);
                    buf_ptr++;
                  }
                  printf ("\n");
                #endif

                after_slope = 0;   /* Record processed and logged: Reset  */
                buf_ptr = buffer;  /* this whole piece of crap and wait   */
                                   /* for new slope to trigger the logger */
              }
             }

             /* Update frequency values of previous loops */

             old_freq1 = old_freq0;
             old_freq0 = frequency;
           }
         }
        #ifdef FSU_DETECTION
          }
        #endif

       #ifdef PROCESS_CHANGES_ONLY
          old_data = data;           /* Save old data for next read process */
        }
       #endif

       data_avail = 0;                /* Data processed: Reset 'data_avail'  */
     }

     /*-- Process user's load control input (overrides automatic control) ---*/

     #ifndef LCD
       if (kbhit ())                    /* Keyboard buffer not empty?        */
       {
         kb_input = getch ();           /* Read key                          */

         #ifndef DATA_LOGGER
           if (kb_input == 'm')         /* Turn load ON unconditionally      */
          {                            /* (user override mode is active)    */
            load_state = load_state | 0x01;

            #ifdef DEBUG
                gotoxy (3, RESULT_OUTPUT_Y - 2);
                printf ("MANUAL MODE   ");
            #endif
          }
          else if ((kb_input == 'a'))  /* Turn load ON if grid frequency is */
               {                       /* OK (GFA control logic activated)  */
                 load_state = load_state & 0xFE;

                 #ifdef DEBUG
                    gotoxy (3, RESULT_OUTPUT_Y - 2);
                    printf ("AUTOMATIC MODE");
                 #endif
               }
         #endif
       }
```

```
    #else
      if (Push_Button_1 () == BUTTON_PRESSED)       /* Evaluate push-button */
      {                                             /* instead of keyboard  */
        while (Push_Button_1 () == BUTTON_PRESSED); /* in LCD mode          */
        if ((load_state == 0) || (load_state == 2))
        {
          load_state = load_state | 0x01;
          LCD_Printf ("Manual Mode   ", 1, 4);
        }
        else
        {
          load_state = load_state & 0xFE;
          LCD_Printf ("Automatic Mode", 1, 4);
        }
      }
    #endif

    /*-- Apply the manual- and automatic load settings ---------------------*/

    #ifndef DATA_LOGGER
      if (load_state != old_lstate)  /* Just do something on a state change */
      {
       if (load_state == 0)                         /* Load is turned off  */
       {
         outp (LPT1_CONTROL_PORT, inp (LPT1_CONTROL_PORT) & LOAD_OFF);

         #ifdef LCD
           LCD_Printf ("Load State is OFF", 1, 3);
         #endif
       }
       else                                         /* Load is turned on   */
       {
         outp (LPT1_CONTROL_PORT, inp (LPT1_CONTROL_PORT) | LOAD_ON);

         #ifdef LCD
           LCD_Printf ("Load State is ON ", 1, 3);
         #endif
       }
      }
      old_lstate = load_state;                       /* Save old load state */
    #endif

    /*-- Display the settings screen if running in LCD support mode --------*/

    #ifdef LCD
      if (Push_Button_2 () == BUTTON_PRESSED) Show_Software_Settings ();
    #endif

/*-- Different abortion conditions, dependant on mode of operation ---------*/

#ifndef DATA_LOGGER
  #ifndef LCD
    } while (kb_input != 27);     /* Wait for user to press ESC key         */
  #else
    } while (Push_Button_3 () == BUTTON_RELEASED);        /* Button 3 = ESC */
  #endif
#else
  #ifdef EVENT_DRIVEN
    } while (kb_input != 27);     /* Wait for user to press ESC key         */
  #else
    /* Wait for user to press ESC key or until automatic program abortion   */
    /* is activated. This is done by comparing the current system time with */
    /* a time stamp recorded at program startup. DO NOT USE the "clock ()"- */
    /* function for this purpose, which counts the 55ms-PC-timer-ticks,     */
    /* because this will mysterically create problems on date changes (the  */
    /* system date will remain on the date the software has been started.   */
    /* This is especially inconvenient for data logging, where the correct  */
    /* time stamps are significant.                                         */

    }  while  ((kb_input  !=  27)  &&  (TIME_TO_RUN  >  difftime  (time  (NULL),
startup_tstamp)));
  #endif
#endif

  /*-- Normal program abortion -----------------------------------------------*/

  /* Turn FSU Hardware power off on program exit */

  outp (LPT1_CONTROL_PORT, inp (LPT1_CONTROL_PORT) | FSU_OFF);
  delay (500);

  /*-- Remove FSU-ISR before exit ---------------------------------------------*/

  Remove_ISR ();

  /*Turn load off on program exit */

  outp (LPT1_CONTROL_PORT, inp (LPT1_CONTROL_PORT) & LOAD_OFF);

  /* Reset printer port to compatibility mode (reconfigures the eight  data */
  /* lines as outputs). Be careful with enabling this code in order to      */
  /* avoid FSU hardware- and PC printer port damage. It is NOT recommended  */
  /* to enable this command if the printer port is not used with other      */
  /* hardware than the FSU.                                                 */

  /*outp (LPT1_CONTROL_PORT, inp (LPT1_CONTROL_PORT) & DISABLE_TRISTATE_MODE);*/

  /* Some screen outputs on program exit */

  #ifndef LCD
    #ifndef DATA_LOGGER
      printf ("\n\n");
    #endif
    ShowCursor ();
  #else
    printf ("\n");
    delay (DISPLAY_DELAY);
    LCD_Printf      ("Program      aborted                              successfully.
", 1, 3);
    delay (DISPLAY_DELAY);
  #endif
  return (0);
}

/*--This is the end of the file "fsu.c" -------------------------------------*/

/****************************************************************************/
```

```
/***************************************************************************/
/*                                                                         */
/* File "fsu-def.c", created on 04/28/2001 by DLO                         */
/*                                                                         */
/* Symbolic constants used in source file "fsu.c"                         */
/*                                                                         */
/* ----------------------------------------------------------------------- */
/*                                                                         */
/* Copyright (C) 2001 by Daniel L. Oedingen, PNNL                         */
/*                                                                         */
/* File last updated on 08/13/2001                                        */
/*                                                                         */
/* NOTE: Any restrictions mentioned in "fsu.c" apply as well for usage of   */
/*       this file.                                                        */
/*                                                                         */
/***************************************************************************/

/*-- User- and application-specific defined Frequency Thresholds -----------*/

/* NOTE: The thresholds are the outer boundaries (device is turned off);    */
/*       the inner boundaries are LOWER_FREQ_THRESHOLD + FREQ_HYSTERESIS    */
/*       and UPPER_FREQ_THRESHOLD - FREQ_HYSTERESIS, respectively.          */

#define UPPER_FREQ_THRESHOLD   60.050   /* The load is turned off at once   */
#define LOWER_FREQ_THRESHOLD   59.950   /* if one of this thresholds is     */
                                        /* exceeded                         */

#define FREQ_HYSTERESIS        0.010    /* hysteresis for turning the load  */
                                        /* back on after switching it off   */

#define WASHOUT_FILTER_THRES   0.010    /* Defines the maximum allowed      */
                                        /* frequency jump between two read  */
                                        /* values in a row on which the     */
                                        /* relay driver output will react.  */

#define SLOPE_DETECTION_THRES  0.025    /* This value has to be exceeded by */
                                        /* the frequency deviation of three */
                                        /* measurements in a row (i.e.      */
                                        /* within 33 ms) if a slope is      */
                                        /* supposed to be detected. NOTE:   */
                                        /* Turn off average value calcula-  */
                                        /* for faster response.             */

#define FIXED_TIMEOUT          5        /* Specifies the fixed (i.e. the    */
                                        /* minimum) time for which the load */
                                        /* will be turned off after a slope */
                                        /* has been detected.               */

#define MAX_RANDOM_TIMEOUT     10       /* Specifies the variable time com- */
                                        /* ponent for the load to be turned */
                                        /* off; the total timeout is the    */
                                        /* sum of 'FIXED_TIMEOUT' and a     */
                                        /* random number between 0 and      */
                                        /* 'MAX_RANDOM_TIMEOUT'. This helps */
                                        /* preventing the GFAs from turning */
                                        /* back on all at the same moment.  */

#define MAX_RANGE_MISS_NO      5        /* Specifies the number of range    */
                                        /* misses in a row which have to be */
                                        /* detected before the load is tur- */
                                        /* ned off.                         */

#define AVG_VALUE_NO           6        /* 1 = no average calculation       */
```

```
                                        /* Reasonable values are < 60 in    */
                                        /* most cases (1 value / second).   */

#define TIME_TO_RUN             3600    /* Please note that this option is  */
                                        /* available in logger mode only.   */
                                        /* Specifies the program's "time to */
                                        /* live" in seconds until it exits  */
                                        /* automatically. This parameter    */
                                        /* does not affect program abortion */
                                        /* by pressing the <ESC> key. Use   */
                                        /* batchfiles for 'continuous' log- */
                                        /* ging.                            */

/****************************************************************************/

/*-- Screen Settings for displaying the result ----------------------------*/

#define RESULT_OUTPUT_X         20      /* Screen column in 80x25 text mode */
#define RESULT_OUTPUT_Y         24      /* Screen row in 80x25 text mode    */

/****************************************************************************/

/*-- FSU-Hardware Parameters ---------------------------------------------*/

#define BOARD_NUMBER            1       /* Apply the following settings for */
                                        /* the corresponding PCB:           */
                                        /* -------------------------------- */
                                        /* -1 = Prototype board (GFA-Box)   */
                                        /*  0 = FSU_PCB#00                  */
                                        /*  1 = FSU_PCB#01                  */
                                        /*  2 = FSU_PCB#02                  */

/* NOTE: This system is designed for a crystal oscillator producing a       */
/*       frequency of exactly 4915200 Hz (i.e. OFFSET is 81920 @ 60.0 Hz).  */
/*       As you can see, one of these parameters has to be corrected to     */
/*       adjust the software, because the crystal oscillator does not pro-  */
/*       duce exactly the frequency it is supposed to.                      */

/* The following lines configure the software for each of the PCBs or the   */
/* prototype (the software needs the exact crystal oscillator "reference"   */
/* frequency in Hz used on the corresponding hardware). This is at the      */
/* moment the only hardware parameter to adjust the software.               */

#if (BOARD_NUMBER == -1)                /* -------------------------------- */
  #define REF_FREQUENCY         4916995 /* Hardware-dependant settings are: */
#endif                                  /* -------------------------------- */
                                        /* FSU_Prototype............4916995 */
#if (BOARD_NUMBER == 0)                 /* FSU_PCB #00..............4916948 */
  #define REF_FREQUENCY         4916948 /* FSU_PCB #01..............4916948 */
#endif                                  /* FSU_PCB #02..............4916948 */
                                        /* -------------------------------- */
#if (BOARD_NUMBER == 1)                 /* Even though all 3 PCBs use the   */
  #define REF_FREQUENCY         4916948 /* same settings, they should be    */
#endif                                  /* selected by each card number.    */

#if (BOARD_NUMBER == 2)
  #define REF_FREQUENCY         4916948
#endif


#define OFFSET                  81920   /* Offset of the 17-Bit-Divisor     */
                                        /* (evaluated by the range hit /    */
                                        /* range miss hardware) to the      */
                                        /* value at 60.000 Hz              */

/***************************************************************************/

/*-- Compiler Settings ----------------------------------------------------*/

#define DATA_LOGGER_             /* Enables the built-in data logger of this  */
                                 /* software if defined, otherwise the pro-   */
                                 /* gram is in "normal" GFA control mode.     */

#define EVENT_DRIVEN_            /* Selects the data logger mode. If defined, */
                                 /* event-driven logging is active, otherwise */
                                 /* the frequency is logged continuously.     */
                                 /* Of course the DATA_LOGGER label must be   */
                                 /* defined as well.                          */

#define DEBUG                    /* If the label DEBUG is defined, the debug  */
                                 /* code of this program (several control     */
                                 /* outputs) is enabled.                      */

#define LCD_                     /* If defined, this causes the software to   */
                                 /* generate ONLY screen outputs for the      */
                                 /* SEETRON serial LCD; otherwise the screen  */
                                 /* output is sent to the standard EGA/VGA    */
                                 /* adapter (i.e. Video Card / CRT), and no   */
                                 /* serial LCD screen output is generated.    */
                                 /* Please note that full LCD support is only */
                                 /* available in GFA Control Logic Mode, but  */
                                 /* NOT in any of the data logger modes.      */

#define SKIPINTRO_               /* If this label is defined, some useful     */
                                 /* information like the software threshold   */
                                 /* settings and stuff will not be displayed. */

#define SKIPTEST                 /* If this label is defined, the ECP/EPP     */
                                 /* port compliance test is disabled. Don't   */
                                 /* forget to activate the high impedance     */
                                 /* mode of the output drivers manually if    */
                                 /* you are sure your port is at least EPP    */
                                 /* compliant.                                */

#define FSU_DETECTION_           /* If this label is defined, an error        */
                                 /* message is displayed if the data read is  */
                                 /* 0xFF (i.e. FSU is probably disconnected). */
                                 /* Undefine it for proper measurements, as   */
                                 /* 0xFF corresponds the count 'OFFSET-1' in  */
                                 /* the 2-complement code used (60.000 Hz +   */
                                 /* one step).                                */

#define PROCESS_CHANGES_ONLY_    /* Sets the software to process input data   */
                                 /* only if 'data' has changed. This option   */
                                 /* prevents you from reading the same value  */
                                 /* for more than one time in a row; be care- */
                                 /* ful with this if you use the averaging    */
                                 /* functions (i.e. AVG_VALUE_NO is > 1).     */
                                 /* Note that this has nothing to do with the */
                                 /* ISR / main loop syncronization using the  */
                                 /* 'data_avail'-variable.                    */

/***************************************************************************/

/*-- Other Definitions ----------------------------------------------------*/
```

```
#define PROG_VERSION           "1.0.1.3 - [DEBUG]"

#define ASM                    asm
#define byte                   unsigned char
#define word                   unsigned int
#define l_int                  unsigned long int

/*-- Special Settings for Data Logger Mode ---------------------------------*/

#ifdef DATA_LOGGER

  #ifdef    DEBUG              /* Avoid any other screen output except from */
    #undef  DEBUG              /* frequency and time stamp data sets        */
  #endif

  #ifndef   SKIPINTRO          /* Do not perform ECP/EPP port compliance    */
    #define SKIPINTRO          /* test in data logger mode                  */
  #endif

  #ifndef   SKIPTEST           /* Do not perform ECP/EPP port compliance    */
    #define SKIPTEST           /* test in data logger mode                  */
  #endif

  #undef    PROG_VERSION       /* Modify program version message            */
  #define   PROG_VERSION "Data Logger V1.5"

  #ifdef    EVENT_DRIVEN       /* LOGGER_MODE is just used to display mode  */
    #define LOGGER_MODE "[EVENT-DRIVEN]"
  #else
    #define LOGGER_MODE "[CONTINUOUS]"
  #endif
#else

  #ifdef    EVENT_DRIVEN       /* Ensure the event-driven mode is off if    */
    #undef  EVENT_DRIVEN       /* the data logger activator is not defined. */
  #endif
  #ifdef    TIME_TO_RUN        /* Ensure that the normal GFA control logic  */
    #undef  TIME_TO_RUN        /* version cannot quit automatically.        */
  #endif
#endif

/***************************************************************************/

/*-- Hardware-I/O Addresses and Commands -----------------------------------*/


#define LPT1_DATA_PORT         0x378    /* R/W port for data I/O            */
#define LPT1_STATUS_PORT       0x379    /* READ ONLY; shows device status   */
#define LPT1_CONTROL_PORT      0x37A    /* R/W port for ECP/EPP control     */

#define INT_MASK_REG           0x21     /* 8259A interrupt mask register    */
#define INT_CMD_REG            0x20     /* Interrupt command register       */

#define LPT1_INT_NO            0x0F     /* IRQ7 for LPT1 corresponds to     */
                                        /* internal type code #15 (0FH)     */

/* NOTE: Use the OR operator (|) for enabling a bit (set bit High), the AND */
/*       operator (&) for disabling a bit (set bit Low). Setting the 8259A  */
/*       interrupt mask register requires the inverse operations described  */
/*       above (all bits active low in this register).                      */

#define FSU_OFF                0x08     /* Used to manipulate Bit 3 in LPT  */
#define FSU_ON                 0xF7     /* control register                 */
```

```
#define LOAD_ON                 0x04     /* Used to manipulate Bit 2 in LPT  */
#define LOAD_OFF                0xFB     /* control register                 */

#define ENABLE_TRISTATE_MODE    0x20     /* Used to manipulate Bit 5 in LPT  */
#define DISABLE_TRISTATE_MODE   0xDF     /* control register                 */

#define ENABLE_LPT1_INT         0x10     /* Used to manipulate Bit 4 in LPT  */
#define DISABLE_LPT1_INT        0xEF     /* control port                     */


#define ENABLE_8259A_IRQ7       0x7F     /* Used to manipulate Bit 7 in the  */
#define DISABLE_8259A_IRQ7      0x80     /* 8259A interrupt mask register    */
#define RESET_8259A             0x20     /* Clears interrupt controller;     */
                                         /* some people call it rather 'EOI' */
                                         /* (End Of Interrupt)               */

#define TEST_BIT_PATTERN        0x55     /* Bit pattern for test purposes    */

#define BUF_SIZE                256      /* Size of several rotating buffers */

/* PLEASE NOTE that any symbolic constant used in driver code for the LC-   */
/*             display (which may optionally be included) can be found in   */
/*             the corresponding source file "fsu-lcd.c".                   */

/*--This is the end of the file "fsu-def.c" --------------------------------*/

/***************************************************************************/
```

```
/*************************************************************************/
/*                                                                       */
/* File "fsu-lcd.c", created on 07/11/2001 by DLO                        */
/*                                                                       */
/* Implementation of subroutines used in source file "fsu.c" for displaying */
/* messages on a SEETRON serial LC-Display (model G12864 V2.0)           */
/*                                                                       */
/* --------------------------------------------------------------------- */
/*                                                                       */
/* Copyright (C) 2001 by Daniel L. Oedingen, PNNL                        */
/*                                                                       */
/* File last updated on 07/23/2001                                       */
/*                                                                       */
/* NOTE: Any restrictions mentioned in "fsu.c" apply as well for usage of   */
/*       this file.                                                      */
/*                                                                       */
/*************************************************************************/

/*-- Symbolic Constants for Serial Port Handling ---------------------------*/

/* Port addresses */
#define COM1_DATA_PORT_REG       0x3F8
#define COM1_BAUDRATE_LSB_REG    0x3F8
#define COM1_BAUDRATE_MSB_REG    0x3F9
#define COM1_LINE_CTRL_REG       0x3FB
#define COM1_MODEM_CTRL_REG      0x3FC
#define COM1_LINE_STATUS_REG     0x3FD

/* 9600 bps */
#define BAUDRATE_LSB             0x0C
#define BAUDRATE_MSB             0x00
#define SET_BIT7_LCR             0x80
#define RESET_BIT7_LCR           0x7F

/* No parity, 1 stop bit, 8 data bits */
#define LCD_SETTINGS             0x03

#define GFA_VERSION              "GFA Version 1.0.1.3 \0"
#define DISPLAY_DELAY            10000

#define BUTTON_PRESSED           1
#define BUTTON_RELEASED          0

/*************************************************************************/

/*-- Variables used only in LCD-Mode ---------------------------------------*/

char timeout_dummy   [4] = "\0";     /* Dummy for screen output conversions */
char frequency_dummy [7] = "\0";     /* Dummy for screen output conversions */

int  message_state = 0;    /* Used to prevent the software from writing the */
                           /* same message to the LCD again and again in    */
                           /* case of no state change occured. This is      */
                           /* necessary because it takes the program about  */
                           /* 42 ms to transmit a 40-byte string, which in  */
                           /* turn causes a loss of at least 2 data acqui-  */
                           /* sition cycles.                                */

                           /* The following table shows the possible values */
                           /* of "message_status" and the corresponding     */
                           /* messages to be just written once:             */

                           /* 0 = No message is being displayed.            */
                           /* 1 = Range Miss detected.                    */
                           /* 2 = Nothing to report...                    */
                           /* 3 = SLOPE: Turning Load off for xxx seconds.  */
                           /* 4 = Under-frequency load shedding.          */
                           /* 5 = Over-frequency load shedding.           */
                           /* 6 = Under-frequency with hysteresis.        */
                           /* 7 = Over-frequency with hysteresis.         */

time_t slope_old = 0;      /* Used for updating the display in case that    */
                           /* slope detection algorithm is retriggered.     */

/*****************************************************************************/

/*-- unsigned int Push_Button_1 (void) -------------------------------------*/

/* Checks if the push-button no. 1 is pressed or not; this function returns */
/* either BUTTON_PRESSED or BUTTON_RELEASED, according to the present state */
/* of the button.                                                           */

/* It evaluates the "Select In"-signal coming in on pin 13 of the DB-25 LPT */
/* connector. This line corresponds to Bit 4 in the LPT1 status register.   */

unsigned int Push_Button_1 (void)
{
  if ((inp (LPT1_STATUS_PORT) & 0x0010) == 0x0010)
  {
    return BUTTON_RELEASED;
  }
  else
  {
    return BUTTON_PRESSED;
  }
}

/*****************************************************************************/

/*-- unsigned int Push_Button_2 (void) -------------------------------------*/

/* Evaluates the "Paper Empty"-signal coming in on pin 12 of the DB-25 LPT  */
/* connector. This line corresponds to Bit 5 in the LPT1 status register.   */

unsigned int Push_Button_2 (void)
{
  if ((inp (LPT1_STATUS_PORT) & 0x0020) == 0x0020)
  {
    return BUTTON_RELEASED;
  }
  else
  {
    return BUTTON_PRESSED;
  }
}

/*****************************************************************************/

/*-- unsigned int Push_Button_3 (void) -------------------------------------*/

/* Evaluates the "Error"-signal coming in on pin 15 of the DB-25 LPT        */
/* connector. This line corresponds to Bit 3 in the LPT1 status register.   */

unsigned int Push_Button_3 (void)
{
  if ((inp (LPT1_STATUS_PORT) & 0x0008) == 0x0008)
  {
    return BUTTON_RELEASED;
  }
  else
  {
    return BUTTON_PRESSED;
  }
}

/****************************************************************************/

/*-- void InitLCD_COM1 (void) ----------------------------------------------*/

/*   Initializes the serial port COM1 for communication with the used LCD.  */
/*   It configures the the parity-, stop bit-, data bit- and baud rate      */
/*   settings.                                                              */

void Init_LCD_COM1 (void)
{
  /* Apply port settings (no parity, 1 stop bit, 8 data bits) */
  outp (COM1_LINE_CTRL_REG, LCD_SETTINGS);

  /* Prepare to set baudrate  */
  outp (COM1_LINE_CTRL_REG, (inp (COM1_LINE_CTRL_REG) | SET_BIT7_LCR));
  outp (COM1_BAUDRATE_LSB_REG, BAUDRATE_LSB);                       /* LSB */
  outp (COM1_BAUDRATE_MSB_REG, BAUDRATE_MSB);                       /* MSB */

  /* Prepare to transmit data */
  outp (COM1_LINE_CTRL_REG, (inp (COM1_LINE_CTRL_REG) & RESET_BIT7_LCR));
  outp (COM1_MODEM_CTRL_REG, (inp (COM1_MODEM_CTRL_REG) | 0x03));

  /* Backlight on */
  outp (COM1_DATA_PORT_REG, 14);
  while ((inp (COM1_LINE_STATUS_REG) & 0x20) == 0);
}

/****************************************************************************/

/*-- void LCD_ClrScr (void) ------------------------------------------------*/

/*   Fills the screen with blanks and sets the cursor to the upper left     */
/*   corner of the screen. Works in both text modes (4- and 8-line mode).   */

void LCD_ClrScr (void)
{
  outp (COM1_DATA_PORT_REG, 12);                           /* Clear Screen */
  while ((inp (COM1_LINE_STATUS_REG) & 0x20) == 0);
}

/****************************************************************************/

/*-- void LCD_Small_Font_Mode (void) ---------------------------------------*/

/*   As the LC-display enters the 4-line mode after power-on, this func-    */
/*   tion has to be called to enter the 8-line mode.                        */

void LCD_Small_Font_Mode (void)
{
  /* Enter small font mode */
  outp (COM1_DATA_PORT_REG, 26);
  while ((inp (COM1_LINE_STATUS_REG) & 0x20) == 0);
  outp (COM1_DATA_PORT_REG, 'F');
  while ((inp (COM1_LINE_STATUS_REG) & 0x20) == 0);
  outp (COM1_DATA_PORT_REG, '1');
  while ((inp (COM1_LINE_STATUS_REG) & 0x20) == 0);
}

/****************************************************************************/

/*-- void LCD_Printf (const char *, unsigned char, unsigned char) ----------*/

/*   Combines the Turbo C fuctions "gotoxy ()" and "printf ()" for use      */
/*   with LC-Displays. Parameters are the string to be displayed as well    */
/*   as the desired x- and  y-position values. The upper left corner of     */
/*   the screen has the coordinates (1, 1).                                 */

void LCD_Printf (const char * string, unsigned char x_position, unsigned char
y_position)
{
  unsigned char i          = 0;
  unsigned char position   = 0;

  position = 20 * (y_position - 1) + x_position - 1;

  outp (COM1_DATA_PORT_REG, 16);
  while ((inp (COM1_LINE_STATUS_REG) & 0x20) == 0);

  outp (COM1_DATA_PORT_REG, position + 64);
  while ((inp (COM1_LINE_STATUS_REG) & 0x20) == 0);

  for (i = 0; i < strlen (string); i++)
  {
    outp (COM1_DATA_PORT_REG, string [i]);
    while ((inp (COM1_LINE_STATUS_REG) & 0x20) == 0);
  }
}

/****************************************************************************/

/*-- void Init_GFA_Screen (void) -------------------------------------------*/

/*   Draws the common parts of the grid-friendly appliance's LCD screen     */
/*   output. Later on, only the changed parts have to be redrawn using the  */
/*   "LCD_Prinf ()" function, the rest may remain the same.                 */

void Init_GFA_Screen (void)
{
  unsigned char line1 [21] = GFA_VERSION;                /* Default GFA     */
  unsigned char line2 [21] = "-------------------\0";   /* output screen   */
  unsigned char line3 [21] = "Load State is OFF   \0";
  unsigned char line4 [21] = "Automatic Mode      \0";
  unsigned char line5 [21] = "Reading f=??.??? Hz \0";
  unsigned char line6 [21] = "-------------------\0";
  unsigned char line7 [21] = "Waiting for valid   \0";
  unsigned char line8 [21] = "frequency value...  \0";

  LCD_ClrScr ();

  LCD_Printf (line1, 1, 1);
  LCD_Printf (line2, 1, 2);
  LCD_Printf (line3, 1, 3);
  LCD_Printf (line4, 1, 4);
  LCD_Printf (line5, 1, 5);
  LCD_Printf (line6, 1, 6);
  LCD_Printf (line7, 1, 7);
  LCD_Printf (line8, 1, 8);
```

```
}

/****************************************************************************/

/*-- void Show_Software_Settings (void) -----------------------------------*/

/* Views the software threshold settings etc. and waits for the user to     */
/* press the PushButton 2 to return to the main screen.                     */

void Show_Software_Settings (void)
{
  unsigned char dummy [4]  = "\0";
  unsigned char line1 [21] = "Software Settings   \0";   /* GFA settings    */
  unsigned char line2 [21] = "--------------------\0";   /* output screen   */
  unsigned char line3 [21] = "Lower tres.__-   mHz\0";
  unsigned char line4 [21] = "Upper tres.__+   mHz\0";
  unsigned char line5 [21] = "Hysteresis____   mHz\0";
  unsigned char line6 [21] = "Washout f.____   mHz\0";
  unsigned char line7 [21] = "Slew rate__    mHz/T\0";
  unsigned char line8 [21] = "Samples/value_____  \0";

  LCD_ClrScr ();

  LCD_Printf (line1, 1, 1);
  LCD_Printf (line2, 1, 2);
  LCD_Printf (line3, 1, 3);
  LCD_Printf (line4, 1, 4);
  LCD_Printf (line5, 1, 5);
  LCD_Printf (line6, 1, 6);
  LCD_Printf (line7, 1, 7);
  LCD_Printf (line8, 1, 8);

  sprintf (dummy, "%2.0f", (60.0 - LOWER_FREQ_THRESHOLD) * 1000);
  LCD_Printf (dummy, 15, 3);
  sprintf (dummy, "%2.0f", (UPPER_FREQ_THRESHOLD - 60.0) * 1000);
  LCD_Printf (dummy, 15, 4);
  sprintf (dummy, "%2.0f", FREQ_HYSTERESIS * 1000);
  LCD_Printf (dummy, 15, 5);
  sprintf (dummy, "%2.0f", WASHOUT_FILTER_THRES * 1000);
  LCD_Printf (dummy, 15, 6);
  sprintf (dummy, "%3.1f", SLOPE_DETECTION_THRES * 500 / AVG_VALUE_NO);
  LCD_Printf (dummy, 12, 7);
  sprintf (dummy, "%2.2d", AVG_VALUE_NO);
  LCD_Printf (dummy, 19, 8);

  while (Push_Button_2 () == BUTTON_PRESSED);
  while (Push_Button_2 () == BUTTON_RELEASED);
  while (Push_Button_2 () == BUTTON_PRESSED);

  LCD_ClrScr ();
  LCD_Printf (GFA_VERSION, 1, 1);
  LCD_Printf ("--------------------", 1, 2);

  if (load_state == 0)
  {
    LCD_Printf ("Load State is OFF   ", 1, 3);
  }
  else
  {
    LCD_Printf ("Load State is ON    ", 1, 3);
  }

  if ((load_state == 1) || (load_state == 3))
  {
    LCD_Printf ("Manual Mode         ", 1, 4);
  }
  else
  {
    LCD_Printf ("Automatic Mode      ", 1, 4);
  }

  LCD_Printf ("Reading f=       Hz ", 1, 5);
  LCD_Printf ("--------------------", 1, 6);

  if (message_state == 1) LCD_Printf ("RANGE MISS detected.                    ", 1,
7);
  if (message_state == 2) LCD_Printf ("Nothing to report...                    ", 1,
7);
  if (message_state == 3)
  {
    LCD_Printf ("SLOPE: Turning load off for ", 1, 7);
    sprintf (timeout_dummy, "%3.3d", timeout);
    LCD_Printf (timeout_dummy, 9, 8);
    LCD_Printf (" seconds.", 12, 8);
  }
  if (message_state == 4) LCD_Printf ("Under-frequency loadshedding active...  ", 1,
7);
  if (message_state == 5) LCD_Printf ("Over-frequency load shedding active...  ", 1,
7);
  if (message_state == 6) LCD_Printf ("Under-frequency     hysteresis range... ", 1,
7);
  if (message_state == 7) LCD_Printf ("Over-frequency      hysteresis range... ", 1,
7);
}

/*--This is the end of the file "fsu-lcd.c" --------------------------------*/

/***************************************************************************/
```

```
/**************************************************************************/
/*                                                                        */
/* File "fsu-sub.c", created on 04/28/2001 by DLO                         */
/*                                                                        */
/* Implementation of subroutines used in source file "fsu.c"              */
/*                                                                        */
/* ---------------------------------------------------------------------- */
/*                                                                        */
/* Copyright (C) 2001 by Daniel L. Oedingen, PNNL                         */
/*                                                                        */
/* File last updated on 08/13/2001                                        */
/*                                                                        */
/* NOTE: Any restrictions mentioned in "fsu.c" apply as well for usage of */
/*       this file.                                                       */
/*                                                                        */
/**************************************************************************/

/*-- forward declaration of interrupt service routine / interrupt vector ---*/

void interrupt (*oldvect) (void);         /* Old interrupt vector           */
void interrupt FSU_ISR    (void);         /* Interrupt prototype            */

/**************************************************************************/

/*-- void HideCursor (void) ------------------------------------------------*/

void HideCursor (void)
{
  ASM MOV AX, 0100H
  ASM MOV CX, 2607H
  ASM INT 10H
}

/**************************************************************************/

/*-- void ShowCursor (void) ------------------------------------------------*/

void ShowCursor (void)
{
  ASM MOV AX, 0100H
  ASM MOV CX, 0506H
  ASM INT 10H
}

/**************************************************************************/

/*-- void DisplayStartupInfo (void) ----------------------------------------*/

/*   This function displays just all these 'useful' information like date   */
/*   and time of compilation, program version, copyrights and the "do and   */
/*   don't list" of this software.                                          */

void DisplayStartupInfo (void)
{
  #ifndef LCD
    printf ("  -----------------------------------------------------------------------
-------");

    #if (BOARD_NUMBER == -1)
      printf ("\n  FSU.EXE           Test-Software for GFA Frequency Sensor Unit:
Prototype");
    #endif
    #if (BOARD_NUMBER == 0)
      printf ("\n  FSU.EXE           Test-Software for GFA Frequency Sensor Unit:
FSU_PCB #00");
    #endif

    #if (BOARD_NUMBER == 1)
      printf ("\n  FSU.EXE           Test-Software for GFA Frequency Sensor Unit:
FSU_PCB #01");
    #endif

    #if (BOARD_NUMBER == 2)
      printf ("\n  FSU.EXE           Test-Software for GFA Frequency Sensor Unit:
FSU_PCB #02");
    #endif

    printf ("\n                   Program Version : %s", PROG_VERSION);
    printf ("\n                   Created on      : %s, %s", __DATE__, __TIME__);
    printf ("\n  ------------------------------------------------------------
---------");
    printf ("\n  USER SETTINGS  : Lower software turn-off threshold...... %5.3f Hz",
LOWER_FREQ_THRESHOLD);
    printf ("\n                   Upper software turn-off threshold...... %5.3f Hz",
UPPER_FREQ_THRESHOLD);
    printf ("\n                     Hysteresis for turning load back on.... %6.1f
mHz", FREQ_HYSTERESIS * 1000);
    printf ("\n                     Washout filter threshold............... %6.1f
mHz", WASHOUT_FILTER_THRES * 1000);
    printf ("\n                     Slope detection slew rate.............. %6.1f
mHz/T", SLOPE_DETECTION_THRES * 500 / AVG_VALUE_NO);

    #if AVG_VALUE_NO == 1
      printf ("\n                   Average value calculation is deactivated.");
    #else
      printf ("\n                    Calculating average values using %d samples per
value.", AVG_VALUE_NO);
    #endif

    printf ("\n  ------------------------------------------------------------
---------");

  #else
    LCD_ClrScr ();
    LCD_Printf (GFA_VERSION, 1, 1);
    LCD_Printf ("--------------------", 1, 2);
  #endif
}

/****************************************************************************/

/*-- void TestPortCompliance (void) ---------------------------------------*/

/*   This function performs a test of the LPT1 parallel port of your PC     */
/*   considering ECP/EPP compliance. This is necessary if you want to use   */
/*   the 8 data lines as bidirectional I/O pins. If the test failed, the    */
/*   entire program is aborted after informing the user. If the test has    */
/*   been performed successful, the user is informed and the PORT REMAINS   */
/*   IN INPUT MODE (i.e. the output driver remains in tristate mode).       */

void TestPortCompliance (void)
{
  #ifndef LCD
    printf ("\n\n  -----------------------------------------------------------
----------");
```

```
    printf ("\n  ECP/EPP TEST   : Please DISCONNECT FSU for Read/Write Test on LPT1.
Your");
    printf ("\n                       parallel port and FSU might be damaged if not
disconnected.");
    printf ("\n\n  USER INFO       : Press <ENTER> if you are ready to test port
compliance.");

    do
    {
      kb_input = getch ();

    } while (kb_input != 13);            /* Wait for user to press RETURN key */

    kb_input = 65;
  #else
    LCD_Printf ("DISCONNECT FSU now  to perform EPP port compliance test,     then
press <ENTER>.                                       ", 1, 3);
    while (Push_Button_3 () == BUTTON_RELEASED);
    while (Push_Button_3 () == BUTTON_PRESSED);
  #endif

  /* Test approach: 1. Try to enable tristate outputs by performing the     */
  /*                   corresponding bit manipulations in the control       */
  /*                   register (first line).                               */
  /*                2. Write specific bit pattern to the data port.         */
  /*                3. Read bit pattern from the data port. If the data     */
  /*                   read is not the same as the written one, the test    */
  /*                   failed (tristate mode could not be initiated).       */

  outp (LPT1_CONTROL_PORT, inp (LPT1_CONTROL_PORT) | ENABLE_TRISTATE_MODE);

  outp (LPT1_DATA_PORT, TEST_BIT_PATTERN);

  if (inp (LPT1_DATA_PORT) == TEST_BIT_PATTERN)  /* Test was NOT successful */
  {
    #ifndef LCD
      printf ("\n\n  TEST RESULT    : Your LPT1 parallel printer port is NOT ECP/EPP
COMPLIANT.");
      printf ("\n                    Thus, you cannot use an FSU in combination with
this PC.");
      printf ("\n\n  USER INFO      : Press <ESC> to quit program.");
      printf ("\n  -----------------------------------------------------------------
-----------");
    #else
      LCD_Printf ("TEST RESULT:        Your LPT1 parallel  printer port is NOT EPP
compliant.                     Press <ESC> to exit.", 1, 3);
    #endif

    #ifndef LCD
      do
      {
        if (kbhit ()) kb_input = getch ();

      } while (kb_input != 27);           /* Wait for user to press ESC key */

      system ("cls");
    #else
      while (Push_Button_3 () == BUTTON_RELEASED);
      printf ("\n");
      LCD_ClrScr ();
      LCD_Printf (GFA_VERSION, 1, 1);
      LCD_Printf ("--------------------", 1, 2);
      LCD_Printf ("Program aborted     successfully.", 1, 3);
    #endif
    exit (0);
  }
  else
  {                                                     /* Test successful  */
    #ifndef LCD
      printf  ("\n\n    TEST  RESULT       :  Your  LPT1  parallel  port  is  ECP/EPP
compliant.");
      printf ("\n  ---------------------------------------------------------------
-----------");
    #else
      LCD_Printf  ("TEST  RESULT:           Your  LPT1  parallel   printer  port  is  EPP
compliant.          ", 1, 3);
      LCD_Printf ("Press <ENTER>.", 1, 8);
      while (Push_Button_3 () == BUTTON_RELEASED);
      while (Push_Button_3 () == BUTTON_PRESSED);
    #endif
  }
}

/****************************************************************************/

/*-- void Install_ISR (void) ----------------------------------------------*/

/* This function installs the FSU interrupt service handler (function       */
/* "Fsu_Isr") on IRQ7 after saving the old interrupt vector table entry.    */
/* Then, the Intel 8259A standard programmable interrupt controller (or     */
/* equivalent) is set properly to enable hardware interrupts on the line    */
/* printer port 1 (LPT1). Finally, it enables the interrupt on LPT1 by      */
/* setting the corresponding Bit 4 in the port control register to 1.       */

void Install_ISR (void)
{
  #ifdef DEBUG
    #ifndef LCD
      printf ("\n\n  ------------------------------------------------------------
-------------");
      printf ("\n  USER INFO      : Installing FSU-ISR on IRQ7 (LPT1)...... ");
    #else
      LCD_Printf ("Installing FSU-ISR  on LPT1....... ", 1, 3);
    #endif
  #endif

  disable ();                             /* Disable maskable interrupts   */
  oldvect = getvect (LPT1_INT_NO);        /* Save old interrupt vector     */
  setvect (LPT1_INT_NO, FSU_ISR);         /* Install FSU interrupt handler */
  enable ();                              /* Enable maskable interrupts    */

  #ifdef DEBUG
    #ifndef LCD
      printf ("done.");
      printf ("\n  USER INFO      : Enabling hardware interrupt on LPT1.... ");
    #else
      LCD_Printf ("done.", 16, 4);
      LCD_Printf ("--------------------", 1, 5);
      LCD_Printf ("Enabling interrupt  on LPT1....... ", 1, 6);
    #endif
  #endif

  /* Set Bit 7 in interrupt mask register of the Intel 8259A interrupt      */
  /* controller to 0 (enables hardware interrupt on PIC)                    */

  disable ();
  outp (INT_MASK_REG, inp (INT_MASK_REG) & ENABLE_8259A_IRQ7);

  /* Read byte from LPT1 control port, set Bit 4 = 1 and write it to        */
  /* control port again (this enables the hardware interrupt on the port)   */

  outp (LPT1_CONTROL_PORT, inp (LPT1_CONTROL_PORT) | ENABLE_LPT1_INT);
  enable ();

  #ifdef DEBUG
    #ifndef LCD
      printf ("done.");
      printf ("\n  -----------------------------------------------------------------
-----------");
    #else
      LCD_Printf ("done.                    ", 16, 7);
      delay (DISPLAY_DELAY);
    #endif
  #endif
}

/****************************************************************************/

/*-- void Remove_ISR (void) -----------------------------------------------*/

/* At first, this function removes the address of the FSU interrupt service */
/* handler from the interrupt vector table entry corresponding to IRQ7      */
/* (LPT1) and restores the previously used interrupt handler. Then, the PIC */
/* is set to disable IRQ7. Finally, the interrupt input (/ackn-line) of the */
/* parrallel port is disabled in the control register of the parallel port  */
/* by setting Bit 4 to 0.                                                   */

void Remove_ISR (void)
{
  #ifdef DEBUG
    #ifndef LCD
      gotoxy (1, RESULT_OUTPUT_Y + 1);
      printf ("\n\n  ----------------------------------------------------------------
------------");
      printf ("\n  USER INFO      : Removing FSU-ISR on IRQ7 (LPT1)........ ");
    #else
      LCD_Printf ("Removing FSU-ISR    on LPT1....... ", 1, 3);
    #endif
  #endif

  disable ();                             /* Disable maskable interrupts  */
  setvect (LPT1_INT_NO, oldvect);         /* Restore old interrupt vector */
  enable ();                              /* Enable maskable interrupts   */

  #ifdef DEBUG
    #ifndef LCD
      printf ("done.");
      printf ("\n  USER INFO      : Disabling hardware interrupt on LPT1... ");
    #else
      LCD_Printf ("done.", 16, 4);
      LCD_Printf ("--------------------", 1, 5);
      LCD_Printf ("Disabling interrupt on LPT1....... ", 1, 6);
    #endif
  #endif

  /* Set Bit 7 in interrupt mask register of the Intel 8259A interrupt      */
  /* controller to 1 (disables hardware interrupt on PIC)                   */

  disable ();
  outp (INT_MASK_REG, inp (INT_MASK_REG) | DISABLE_8259A_IRQ7);

  /* Read byte from LPT1 control port, set Bit 4 = 0 and write it to        */
  /* control port again (this disables the hardware interrupt)              */

  outp (LPT1_CONTROL_PORT, inp (LPT1_CONTROL_PORT) & DISABLE_LPT1_INT);
  enable ();

  #ifdef DEBUG
    #ifndef LCD
      printf ("done.");
      printf ("\n  -----------------------------------------------------------------
-----------");
    #else
      LCD_Printf ("done.                    ", 16, 7);
    #endif
  #endif
}

/****************************************************************************/

/*-- void interrupt FSU_ISR (void) -----------------------------------------*/

/* This function is the hardware interrupt handler itself. It just reads    */
/* the data byte, writes it to the corresponding global variable and resets */
/* the 8259A PIC to enable further interrupts.                              */

void interrupt FSU_ISR (void)
{
  data = inp (LPT1_DATA_PORT);             /* Read data from parallel port */
  data_avail = 1;                          /* Triggers the main program to */
                                           /* update the frequency value   */
  outp (INT_CMD_REG, RESET_8259A);         /* Reset 8259A after interrupt  */
}

/*--This is the end of the file "fsu-sub.c" --------------------------------*/

/****************************************************************************/
```

```
/***************************************************************************/
/*                                                                         */
/* File "fsu-var.c", created on 04/28/2001 by DLO                          */
/*                                                                         */
/* Global variables used in source file "fsu.c"                            */
/*                                                                         */
/* ----------------------------------------------------------------------- */
/*                                                                         */
/* Copyright (C) 2001 by Daniel L. Oedingen, PNNL                          */
/*                                                                         */
/* File last updated on 07/25/2001                                         */
/*                                                                         */
/* NOTE: Any restrictions mentioned in "fsu.c" apply as well for usage of  */
/*       this file.                                                        */
/*                                                                         */
/***************************************************************************/

/*-- User-defined type definitions ---------------------------------------*/

/* One data set consists of the frequency in float format, the standard C   */
/* time stamp in seconds since 01/01/1970 and, for better accuracy, the     */
/* "hundredth" of a second (i.e. the 55 ms intervals from the system timer  */
/* tick are counted). As a better time stamp is not easy to get from the PC */
/* hardware / OS, evaluating this part of the timestamp is only reasonable  */
/* for averaged values ('AVG_VALUE_NO' should be at least >= 4...6).        */

struct dataset { float dataset_frequency;
               l_int dataset_timestamp_sec;
               byte  dataset_timestamp_hund;
              };

/*-- Declaration of global variables -------------------------------------*/

byte  kb_input     = 0;   /* Contains characters read from the keyboard     */

byte  load_state   = 0;   /* If 0 or 2, the GFA logic is enabled to control */
                      /* the load (0 = OFF). State 1 or 3 means that    */
                      /* the user overrides the GFA control logic (de-  */
                      /* vice is permanently ON). Bit 0 represents the  */
                      /* user's settings, Bit 1 settings are applied    */
                      /* automatically by the GFA control logic.        */

byte   old_lstate  = 0;   /* Contains load status of last loop cycle        */

byte   data        = 0;   /* Contains data byte read from parallel port     */

byte   data_avail  = 0;   /* Synchronizes main program loop with ISR        */

byte   startup     = 1;   /* Helps initializing the GFA control logic       */
                      /* (prevents problems during program startup and  */
                      /* FSU hardware power-on). It is set to zero when */
                      /* the first valid frequency value has bee read   */
                      /* (i.e. a value within the software thresholds). */

byte   range_miss = 0;    /* Counts the number of range misses in a row.    */
                          /* This variable is used to prevent the software  */
                          /* from turning off the load briefly if a single  */
                          /* (or just a few) range miss-messages are read.  */

#ifdef PROCESS_CHANGES_ONLY
  byte old_data    = 0;   /* Contains data of last read process             */
#endif
```

```
#ifdef EVENT_DRIVEN
  word event_count = 0;   /* Counts the number of detected events           */
#else
  #ifdef DATA_LOGGER
    time_t startup_tstamp = 0;
                          /* Contains time stamp at which the program has   */
                          /* been started; used for automatic program       */
                          /* abortion in continuous data logger mode.       */
  #endif
#endif

word   timeout     = 0;   /* Represents the fixed + the randomized timeouts */

time_t sample_time = 0;   /* Contains the time in seconds for a sample      */

time_t slope_time  = 0;   /* Time in seconds since 01/01/1970 at the moment */
                      /* a slope has been detected                      */

word   avg_count   = 0;   /* Just a counter variable for display purposes   */

word   after_slope = 0;   /* Controls data reading process after an event   */
                      /* (e.g. a slope) occured; 0 = FALSE, otherwise = */
                      /* TRUE ('after_slope' is also used as a loop     */
                      /* counter for output purposes etc.)              */

l_int  divisor     = 1;   /* Divisor to determine grid frequency            */

float  frequency   = 0.0; /* Grid frequency read in float format ("loop n") */

float  old_freq0   = 1.0; /* Contains frequency in loop n-1                 */

float  old_freq1   = 2.0; /* Contains frequency in loop n-2                 */

double freq_sum    = 0.0; /* Contains sum of frequency values read since    */
                      /* last clear if average values are used          */

struct dataset buffer [BUF_SIZE];  /* Rotating memory for event-driven data */
                             /* logging and Fast Fourier Transform;   */
                             /* contains the last <BUF_SIZE> pro-     */
                             /* cessed frequency values and time      */
                             /* stamps                                */

struct dataset * buf_ptr   = NULL; /* This pointer is used for any I/O      */
                             /* operation of the 'buffer [BUF_SIZE]'  */
                             /* rotating memory                       */

struct time t;            /* Necessary to get the hundredth of a second     */
struct date d;            /* Contains the present date                      */

/* PLEASE NOTE that any variable used in driver code for the LC-display     */
/*             (which may optionally be included) can be found in the       */
/*             corresponding source file "fsu-lcd.c".                       */

/*--This is the end of the file "fsu-var.c" --------------------------------*/

/***************************************************************************/
```

## Analysis and Controls Software for the Data Analysis Platform

The controls software consists of the following modules in ANSI-C language[7].

- gfa.c
- gfa_dsp.c
- gfa_fft.c
- gfa_if.c
- gfa_server.c

Each module is listed below.

[7] ANSI, 1988. Draft Proposed American National Standard for Information Systems -- Programming Language C. Technical Report X3J11/88-158, ANSI Accredited Standards Committee, X3 Information Processing Systems, December.

```
/*************************************
*      File    : gfa.c
*      Project : Grid Friendly Appliance
*      Author  : Steffen Lang, PNNL
*
*      control and analyse on the signal
*
*************************************/

/* include files */
#include "gfa.h"
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <termios.h>
#include <signal.h>
#include <unistd.h>
#include <sys/time.h>
#include <fcntl.h>
#include <sys/stat.h> /* for mode definitions */

/* glob vars */
unsigned int arr[60];
FILE * hfp = NULL;

static int histogram   = 0;
static int page_select = 1;
static int peek        = -1;
static struct termios orig_term, new_term;
static unsigned char load_state = 0x00;
#ifdef PROTOCOL
static FILE *fp_protocol = NULL;
#endif

int    new_data_avail = 0;
int    end_of_file = 0;
int    analysis_running = 0;
int    operation_mode = 0;
int    logfile_type;
char   log_time[100];
server_state state = NO_SOCKET;

double samples [NUM_OF_SAMPLES];
double spectrum[NUM_OF_SAMPLES];
double diff1[NUM_OF_SAMPLES];
double diff2[NUM_OF_SAMPLES];
double imOut[NUM_OF_SAMPLES];
double reOut[NUM_OF_SAMPLES];
double stddevfft;
int    zero_crossings1[NUM_OF_SAMPLES];
int    zero_crossings2[NUM_OF_SAMPLES];

double threshold[5]  = {-120.0, -120.0, -120.0, -120.0, -120.0};
double thres_integral[5];
#define MAX_BUF 40
double ibuf[MAX_BUF][5];
int    ibufpos  = 0;
int    ibufinit = 0;

PEAK   peaks[NUM_OF_SAMPLES/2];
```

```
/* function prototypes */
void help(void);
int  time_analyse(double samples[]);
int  fft_analysis(void);
int  console_display();
int  readch();
int  kbhit();
int  init(int argc, char *argv[]);
int  histogram_handler(void);
int  set_relay(int relay, int state);

/*********************************************
*      main()
*      -----------------------------------------
*
*********************************************/
int main (int argc, char *argv[])
{
       int ch;

       #ifdef PROTOCOL
       char * protocol_file  = "protocol.txt";
       if((fp_protocol=fopen(protocol_file,"w"))==NULL)
       {
              printf("ERROR : Can not open '%s'\n",protocol_file);
              exit(0);
       }
       #endif

       init(argc,argv);

       while(1)
       {
              //checking if keyboard was hit
              if(kbhit())
              {
              ch = readch();
              if(ch=='q')         //quit the program
              {
                     break;
              }
              else if(ch>='0' && ch<='9')
              {
                     if(ch=='0')
                            set_relay(0,0);            //AUS
                     else if(ch=='1')
                            set_relay(0,-1);    //AN
                     page_select=ch-'0';
              }
       }

       if(end_of_file)            //if reading from logfile and end of file reached
       {
              int i;
              printf("\nEnd of file reached\n");

                     /*     this is for occurence
              for(i=0;i<50;i++)
              {
                     fprintf(hfp," %d %d \n",i,arr[i] );
              }
              */
              fclose(hfp);
```

```
            break;
        }

            if(histogram)       //call  explicit  get_samples  -  otherwise  it  is
called
            {                               //by the timer
                get_samples(SIGALRM);
            }

            if(new_data_avail)
            {
                analysis_running = 1;           //flag

                /*+++++++++++ analysis in time domain ++++++++++++++++++++*/
                //time_analyse(samples);
                #ifdef PROTOCOL
                {
                    int t;
                    fprintf(fp_protocol,"\n\ntime signal\n");
                    fprintf(fp_protocol,"-----------\n");
                    for(t=0;t<=255;t++)
                    {
                        fprintf(fp_protocol,"%7.4f ",samples[t]);
                    }
                }
                #endif

                /*+++++++++++      analysis      in      frequency      domain
++++++++++++++++++++*/
                power_fft(samples,spectrum);
                fft_analysis();
                #ifdef PROTOCOL
                {
                    char cbuffer[100];
                    fprintf(fp_protocol,"\n\nfrequency signal\n");
                    fprintf(fp_protocol,"----------------\n");
                    for(t=0;t<=255;t++)
                    {
                        sprintf(cbuffer,"\n%03d  %09.5f",t,spectrum[t]);
                        *(strchr(cbuffer,'.'))=',';             //if  excel
format
                        fprintf(fp_protocol,cbuffer);
                        #ifdef FFT_IN_DB
                            fprintf(fp_protocol,"\tdb");
                        #endif
                    }
                }
                #endif

                analysis_running = 0;
                new_data_avail = 0;

                if(histogram)
                {
                    histogram_handler();
                }
                else
                {
                    server_handler();
                    //console_display();
                }
            }
        }
```

```
	#ifdef PROTOCOL
		fclose(fp_protocol);
	#endif

	tcsetattr(0,TCSANOW, &orig_term);
    return 1;
}

/*******************************************
*	init()
*	---------------------------------------
*
*******************************************/
int init(int argc, char *argv[])
{
	set_relay(0,-1);		//turn load on
	init_interface(argc, argv);
	init_fft();

	if(argc==4 && (strcmp(argv[3],"-r")==0) )
	{
		histogram = 1;
	}

	//clear the screen
	clrscr();
	//init kbhit
	tcgetattr(0, &orig_term);
	new_term = orig_term;
	new_term.c_lflag &= ~ICANON;
	new_term.c_lflag &= ~ECHO;
	new_term.c_lflag &= ~ISIG;
	new_term.c_cc[VMIN] = 1;
	new_term.c_cc[VTIME] = 0;
	tcsetattr(0, TCSANOW, &new_term);

	return 0;
}

/*******************************************
*	fft_analysis()
*	---------------------------------------
*
*******************************************/
int fft_analysis(void)
{
	int i;
	int n;

	//standard deviation fft
	stddevfft = stddev(spectrum+3,NUM_OF_SAMPLES-4);

	//differentation
	diff(spectrum,diff1,zero_crossings1,50);
	diff(diff1,diff2,zero_crossings2,50);

	//integral
	for(i=0;i<5;i++)
	{
		t_integral(spectrum+(i*10),threshold[i],&thres_integral[i],10);
		ibuf[ibufpos][i] = thres_integral[i];	//update ibuf
		//if(i==1)
		//printf("%d %d %f\n",ibufpos,i,ibuf[ibufpos][i]);
}

//position ptr
ibufpos = (ibufpos+1)%MAX_BUF;
if(!ibufinit && !ibufpos)
        ibufinit = 1;

if(ibufinit)        //enough integrals have been calculated
{
       #define BAND_X      2            //Band 1-5
       int i;
       double av  = 0.0;
       double std = 0.0;          //stddev of integral in BAND
       static double avx  = 0.0;
       static double stdx = 0.0;

       int out_of_range = 0;

       for(i=0;i<MAX_BUF-10;i++)  //get average value
       {
             av+=ibuf[(ibufpos+i)%MAX_BUF][BAND_X-1];
       }
       av/=MAX_BUF-10;

       for(i=0;i<MAX_BUF-10;i++)  //get standard deviation
       {
             std+=pow((ibuf[(ibufpos+i)%MAX_BUF][BAND_X-1]-av),2);
       }
       std = sqrt(std/(MAX_BUF-10-1));
       //printf("std : %f\n",std);

       for(;i<MAX_BUF;i++)
       {
             if(ibuf[(ibufpos+i)%MAX_BUF][1] > av+(2.0*std))
             {
                   out_of_range++;
             }
       }

       if(load_state==0x01)
       {
             //2 criteria for turning the load again on
             if(std<=stdx || av<=avx)
             {
                   set_relay(0,-1);           //turn on
                   load_state=0x00;
             }
       }

       if(out_of_range>=10)
       {
             if(load_state==0x00)       //is on
             {
                   set_relay(0,0);            //turn off
                   avx  = av;
                   stdx = std;
                   load_state=0x01;
             }
       }

       printf("out: %d, stddev: %f, av: %f",out_of_range,std,av);
       printf("--- %s  ---\n",log_time);
}
```

```
	/*		//integral calculation with different defintion of band 1 and 2
	{
		#define THRESHOLD -120.0
		#define BAND1_START	02
		#define BAND1_END	13
		#define BAND2_START	13
		#define BAND2_END	24

		t_integral(spectrum+BAND1_START,THRESHOLD,	&thres_integral[0],
BAND1_END-BAND1_START+1);
		t_integral(spectrum+BAND2_START,THRESHOLD,	&thres_integral[1],
BAND2_END-BAND2_START+1);
		thres_integral[2]=0.0;
		thres_integral[3]=0.0;
		thres_integral[4]=0.0;
	}*/

	//peak detection
	i=0;
	n=0;
	while(zero_crossings1[i] != -1)
	{
		if(diff2[zero_crossings1[i]-1]<0.0)
		{
			double angle,angle1,angle2;

			//calculation
			angle1	=	(atan(DIAGRAM_RELATION/diff1[zero_crossings1[i]-
1])/PI)*180;
			angle2							=
(atan(DIAGRAM_RELATION/diff1[zero_crossings1[i]])/PI)*180;
			angle  = angle1-angle2;

			peaks[n].pos   = zero_crossings1[i];
			peaks[n].value = spectrum[zero_crossings1[i]];
			peaks[n].angle = angle;
			peaks[n].sharp = diff2[zero_crossings1[i]-1];
			n++;
		}
		i++;
	}
	peaks[n].pos = -1;	//mark the end

	return 0;
}

/*********************************************
*	set_relay()
*	---------------------------------------
*	parameter:
*	 relay   0 ... 5
*	 state   -1,0,1
*
*	return :
*	 0 OK
*	 1 ERROR
*********************************************/
int set_relay(int relay, int state)
{
	FILE * fp;
```

```
        char file[100]= "/proc/dx2/control/";
        char *files[] = {"fanr","clgr","htgr","revc","defr","lckr"};

        if(relay<0 || relay>5 || state<-1 || state>1)
        {
               //printf("Error2 in set_relay\n");
               return 1;
        }
        strcat(file,files[relay]);
        if((fp=fopen(file,"wt"))==NULL)
        {
               //printf("Error3 in set_relay (opening %s)\n",file);
               return 1;
        }
        //printf("%d\n",state);
        if(fprintf(fp,"%d\n",state)<0)
        {
               return 1;
        }
        fclose(fp);
        return 0;
}


/*********************************************
*       help()
*       -----------------------------------------
*
*********************************************/
void help(void)
{
        printf("**************************************************************\n");
        printf("*  Call with : gfa [file date [-r]] \n");
        printf("*   No optional parameters -> receive current frequency from the grid
\n");
        printf("*    Optional parameters     -> receive frequency from the log file
\n");
        printf("*   ---------------------------------------------------------\n");
        printf("*   file : path of log file (*.conv, *.fsu)\n");
        printf("*   date : date and time of start of processing\n");
        printf("*          format mm/dd/yy/hh/mm/ss \n");
        printf("*   -r   : histogram mode");
        printf("*          \n");
        printf("**************************************************************\n");
        return;
}

/*********************************************
*       console_display()
*       -----------------------------------------
*       displaying information on the current
*       analysis
*       display mode is determined by global
*       variable page_select
*
*       parameter :
*     none
*
*       return :
*        always 0
*
**********************************************/
int console_display()
```

```
{
	int i;
	char str_server[2];

	str_server[0] = '-';
	str_server[1] = 0;

	if(state == CLIENT)
	{
		str_server[0] = 'C';
	}
	else if(state==REQUEST)
	{
		str_server[0] = 'T';
	}

	clrscr();
	if(page_select==1)
	{
		printf("(1)  Grid  Frequency  -  Timer  Interval:  %d  sec,  Network:
%s\n",TIMER_INTERVAL, str_server);
		if(operation_mode==REALTIME)
		{
			printf("-------------------------------------------------------
--------------------\n");
		}
		else
		{
			printf("----- %s  -----\n",log_time);
		}

		for(i=0;i<20;i++)
		{
			int y;
			for(y=0;y<120;y+=20)
			{
				printf("%03d ",i+y+1);
				printf("\033[1;34m");
				printf("%7.4f  ",samples[i+y]);
				printf("\033[0m");
			}
			printf("\n");
		}
	}
	else if(page_select==2)
	{
		printf("(2)  Grid  Frequency  -  Timer  Interval:  %d  sec,  Network:
%s\n",TIMER_INTERVAL, str_server);
		if(operation_mode==REALTIME)
		{
			printf("-------------------------------------------------------
--------------------\n");
		}
		else
		{
			printf("----- %s  -----\n",log_time);
		}

		for(i=0;i<20;i++)
		{
			int y;
			for(y=120;y<240;y+=20)
			{
```

```
                              printf("%03d ",i+y+1);
                              printf("\033[1;34m");
                              printf("%7.4f  ",samples[i+y]);
                              printf("\033[0m");
                        }
                        printf("\n");
                  }
            }
            else if(page_select==3)
            {
                  printf("(3)  FFT  Spectrum      -  Timer  Interval:  %d  sec,  Network:
%s\n",TIMER_INTERVAL, str_server);
                  if(operation_mode==REALTIME)
                  {
                        printf("-----------------------------------------------------
--------------------\n");
                  }
                  else
                  {
                        printf("----- %s  -----\n",log_time);
                  }

                  for(i=0;i<=20;i++)
                  {
                        printf(" %5.3f Hz  ",(float)i*0.039);
                        printf("\033[1;31m");
                        printf("%8.4f db\t   ",spectrum[i]);
                        printf("\033[0m");
                        printf(" |\t    %5.3f Hz  ",(float)(i+21)*0.039);
                        printf("\033[1;31m");
                        printf("%8.4f db \n",spectrum[i+21]);
                        printf("\033[0m");
                  }
            }
            else if(page_select==4)
            {
                  int u;
                  printf("(4)  Analysis            -  Timer  Interval:  %d  sec,  Network:
%s\n",TIMER_INTERVAL, str_server);

                  if(operation_mode==REALTIME)
                  {
                        printf("-----------------------------------------------------
--------------------\n");
                  }
                  else
                  {
                        printf("----- %s  -----\n",log_time);
                  }
                  printf(" Band  Threshold   Integral\t\tStandard Deviation : ");
                  printf("\033[1;35m");
                  printf("%7.4f\n",stddevfft);
                  printf("\033[0m");

                  #ifdef PROTOCOL
                  fprintf(fp_protocol," Band\t    Threshold \t Integral\n");
                  #endif
                  for(i=0;i<5;i++)
                  {
                        printf("  %02d   %08.3f ",i+1,threshold[i]);
                        printf("\033[1;35m");
                        printf("   %08.5f\n",thres_integral[i]);
                        printf("\033[0m");
```

```
			#ifdef PROTOCOL
			fprintf(fp_protocol,"%d band:   %08.3f ",i+1,threshold[i]);
			fprintf(fp_protocol,"\t %08.5f\n",thres_integral[i]);
			#endif
		}

		printf("\n Peak\t Value\t\tAngle\t  Sharpness (2.dev)\n");
		i = 0;
		u = 0;
		while(peaks[i].pos >=0)
		{
			//printf("                    %02d\t%05.2f          db\t%4.2f°\t
%5.2f\n",zero_crossings1[i],spectrum[zero_crossings1[i]],angle1-
angle2,diff2[zero_crossings1[i]-1]);
			printf("                    %02d\t%05.2f           db\t%4.2f°\t
%5.2f\n",peaks[i].pos,peaks[i].value,peaks[i].angle,peaks[i].sharp);
			#ifdef PROTOCOL
			//fprintf(fp_protocol,"peak  at  %02d  \t  value  %f  sharpness
%f\n",zero_crossings1[i],spectrum[zero_crossings1[i]],diff2[zero_crossings1[i]-1]);
			#endif
			i++;
		}
		#ifdef PROTOCOL
		fprintf(fp_protocol,"\n");
		#endif
	}
	else
	{
		printf("(%d) no valid mode\n",page_select);
		printf("-----------------------------------------------------------
--------------\n");
	}
	return 0;
}

/*********************************************
*	histogram_handler()
*	---------------------------------------
*
*	parameter :
*    none
*
*	return :
*	 0 OK
*	 1 ERROR
*
*********************************************/
int histogram_handler(void)
{
	static int is_init = 1;
	//static FILE * hfp = NULL;
	static unsigned int count = 1;
	int i;
	int n;
	int max[3];

	//open file if first call
	if(is_init)
	{
		if((hfp=fopen("histogram.txt","w"))==NULL)
		{
			printf("Error in open histogram.txt\n");
```

```
                return 1;
            }
            printf("Grid Friendly Appliance - Histogram mode\n");
            printf("----------------------------------------\n");
            printf("\033[4;3H");              //set cursor to (1,1)
            printf("Samples    Date+Time");

            fprintf(hfp,"d5 thres integrals\n\n");
            fprintf(hfp," band1     band2      band3      band4      band5\n");
            fprintf(hfp,"-----------------------------------------------------------
-------------------------------\n");
            is_init = 0;
        }

        //print to file
        //fprintf(hfp,"%7.4f  ",stddevfft);

        for(i=0;i<5;i++)
        {
            char str[50];
            sprintf(str," %05.3f",thres_integral[i]);
            *(strchr(str,'.'))=',';           //excel format
            fprintf(hfp,str);
            fprintf(hfp,"  |  ");
        }

        /*
        for(n=1;n<=5;n++)
        {
            char str[50];
            get_band_max(n,max);
            i=0;

            if(max[0]!=-1)
            {
                //printf(" band%d - increase pos %d",n ,max[i]);
                //getchar();
                arr[peaks[max[i]].pos]++;
            }
        }*/

        /*
        for(n=1;n<=5;n++)
        {
            char str[50];
            get_band_max(n,max);
            i=0;
            while(i<1)
            {
                if(max[i]!=-1)
                {
                    if(peaks[max[i]].angle<100.00)
                        sprintf(str,"%05.3f ",peaks[max[i]].angle);
                        //sprintf(str,"%05.1f ",peaks[max[i]].sharp);
                    else
                        sprintf(str,"%05.2f ",peaks[max[i]].angle);
                        //sprintf(str,"%05.2f ",peaks[max[i]].sharp);

                    *(strchr(str,'.'))=',';           //excel format
                    fprintf(hfp,str);
                }
                else
                {
```

```
                                fprintf(hfp,"xxxxxx ");
                                //fprintf(hfp,"%02d ",0);
                        }
                        i++;
                }
                fprintf(hfp," | ");
        }*/


        {
                char string[20];
                strncpy(string,log_time+11,8);
                string[8] = '\0';
                fprintf(hfp,"%s\n",string);             //print time to histogram file
        }

        //print time on screen
        printf("\033[5;3H");            //set cursor to (1,1)
        printf("%05d      %s \n",count++, log_time);

        return 0;
}


/*********** kbhit emulation *****************/
int kbhit()
{
        char ch;
        int nread;

        if(peek != -1)
        {
                return 1;
        }

        new_term.c_cc[VMIN]=0;
        tcsetattr(0, TCSANOW, &new_term);
        nread = read(0,&ch,1);
        new_term.c_cc[VMIN]=1;
        tcsetattr(0, TCSANOW, &new_term);

        if(nread == 1)
        {
                peek = ch;
                return 1;
        }
        return 0;
}

int readch()
{
        char ch;

        if(peek != -1)
        {
                ch = peek;
                peek = -1;
                return ch;
        }
        read(0,&ch,1);
        return ch;
}
```

```
/*************************************
*      File    : gfa_dsp.c
*      Project : Grid Friendly Appliance
*      Author  : Steffen Lang, PNNL
*
*      math. functions for 'Digital Signal Processing'
*
*************************************/

#include "gfa.h"
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>

extern PEAK   peaks[NUM_OF_SAMPLES/2];

/*******************************************
*      get_band_max()
*      ----------------------------------------
*      detects the 3 maximum values values in one
*   band
*
*      parameter :
*       band    - indicates the concerned band (1..5)
*       index   - array where the results are stored
*
*      return :
*       0 = OK
*       1 = ERROR
*
*******************************************/
int get_band_max(int band, int index[])
{
      int i     = 0;
      int upper = 9 + (band-1)*10;
      int lower = 0 + (band-1)*10;

      index[0]=index[1]=index[2]=-1;

      while(peaks[i].pos!=-1)
      {
            if(peaks[i].pos>=lower && peaks[i].pos<=upper)       //if peak in band
            {
                  int n = 0;
                  int empty = 0;

                  for(n=0;n<3;n++)
                  {
                        if(index[n]==-1)
                        {
                              empty = 1;
                              break;
                        }
                        else if(peaks[i].value > peaks[index[n]].value)
                        {
                              break;
                        }
                  }

                  if(empty)
```

```
                    {
                           index[n] = i;
                    }
                    else if(n<3)
                    {
                           switch(n)
                           {
                                  case 2:
                                         index[2] = i;
                                         break;
                                  case 1:
                                         index[2] = index[1];
                                         index[1] = i;
                                         break;
                                  case 0:
                                         index[2] = index[1];
                                         index[1] = index[0];
                                         index[0] = i;
                                         break;
                           }
                    }
             }
             i++;
      }

      return 0;
}

/*********************************************
*      detect_extremum()
*      ----------------------------------------
*      detects extremum values (max and min)
*   in the signal
*
*      parameter :
        sig   - signal to be analyzed
*       max    - 1 = max detection
*                      0 = min detection
*       start - start index in signal
*       end   - end index in signal
*
*      return :
*       the index of the max value in
*       the data array
*
*********************************************/
int detect_extremum(double sig[], int max, const int start, const int end)
{
      int i;
      double extrem_value;
      int extrem_position;

      extrem_value    = sig[start];
      extrem_position = start;

      for(i=start;i<end;i++)
      {
             if(max)                     // max detection
             {
                    if(sig[i]>extrem_value)
                    {
                           extrem_value    = sig[i];
                           extrem_position = i;
```

```
                        }
                }
                else                    // min detection
                {
                        if(sig[i]<extrem_value)
                        {
                                extrem_value    = sig[i];
                                extrem_position = i;
                        }
                }
        }
        return extrem_position;
}

/**********************************************
*       stddev() - proved with Matlab (std function)
*       -----------------------------------------
*       calculates the standard deviation
*
*       parameter :
*     sig  - samples of the signal
*     num  - length of signal
*
*       return :
*        standard deviation
*
**********************************************/
double stddev(double sig[], const int len)
{
        int i;
        double average=0.0;
        double temp   =0.0;

        for(i=0;i<len;i++)
        {
                average+=sig[i];
        }
        average/=len;
        for(i=0;i<len;i++)
        {
                temp+=pow((sig[i]-average),2);
        }
        return sqrt(temp/(len-1));
}

/**********************************************
*       diff()  -  proved with Matlab
*       -----------------------------------------
*       derivation of a signal and calculation
*       of the positions of the zero-crossings
*
*       parameter :                                             (length)
*     sig  -  original signal      f(x)          n
*     ddt  - derived  signal      df(x)/dt       n-1
*     zc   - zero crossings in ddt         n
*        len  - lenght of ori. signal
*
*       return :
*        always 0
*
**********************************************/
#define POS   1
#define ZERO  0
```

```
#define NEG  -1
int diff(double sig[], double ddt[], int zc[], const int len)
{
       int i;
       int current_sign;
       int previous_sign = ZERO;
       int index = 0;

       for(i=0;i<len-1;i++)
       {
              ddt[i] = (sig[i+1]-sig[i]);              //diff

              if(ddt[i]!=0.00)                         //zero crossings
              {
                     current_sign = (ddt[i] > 0.0) ? POS : NEG;
                     if(previous_sign!=ZERO &&
                        current_sign!=previous_sign)
                     {
                            zc[index++] = i;    /* save 2. index of zerocrossing */
                     }
              }
              else
                     current_sign = ZERO;

              previous_sign = current_sign;
       }
       zc[index]= -1;             //mark the end
       return 0;
}

/*********************************************
*      integration()
*      ---------------------------------------
*      performs the integration on the given
*      signal
*
*      parameter :                             (length)
*    sig    - original signal           n
*    integr - integration
*       len    - length of ori. sig.
*
*      return :
*       always 0
*
*********************************************/
void integration(double sig[], double integr[], const int len)
{
       int i;
       double temp;

       integr[0] = 0;

       for(i=0;i<(len-1);i++)
       {
              temp = (sig[i]+sig[i+1]/2)*0.1;
              integr[i+1] = integr[i] + temp;
       }
       return;
}

/*********************************************
*      t_integral()
*      ---------------------------------------
*      performs the integration on the given
*      signal
*
*      parameter
*    sig    - original signal
*       thres  - threshold
*    integr - integration
*       len    - length of original signal
*
*      return :
*       always 0
*
*********************************************/
void t_integral(double sig[], double thres, double * integral, const int len)
{
       int i;
       double temp;
       double d1,d2;

       *integral = 0;

       for(i=0;i<(len-1);i++)
       {
              d1 = sig[i]-thres;
              d2 = sig[i+1]-thres;
              if( d1>=0.0 && d2>=0.0 )
              {
                     temp = ((d1+d2)/2)*0.1;
                     (*integral)+= temp;
              }
       }
       return;
}

/*********************************************
*      xcorr()  - proved with Matlab
*      ----------------------------------------
*      performs the crosscorrelation of two
*      signals. The result is written into array
*      xcorr
*
*      parameter :                             (length)
*    sig1  - signal 1                         n
*    sig2  - signal 2                         n
*    xcorr - cross correlation          2*n-1
*
*      return :
*       always 0
*
*********************************************/
int xcorr(double sig1[], double sig2[], double xcorr[], const int len)
{
       int i;              //index outer loop
       int ii;                    //index inner loop
       int index_sig1;
       int index_sig2;
       double temp;

       for(i=-len+1;i<len;i++)           //outer loop
       {
              temp = 0;
              if(i<=0)
              {
```

```
                        index_sig1 = 0;
                        index_sig2 = abs(i);
                }
                else
                {
                        index_sig1 = i;
                        index_sig2 = 0;
                }

                for(ii=0;ii<(len-abs(i));ii++)          //inner loop
                {
                        temp+=sig1[index_sig1++]*sig2[index_sig2++];
                }
                xcorr[i-1+len]= temp;
        }
        return 0;
}
```

```
/*************************************
*      File    : gfa_fft.c
*      Project : Grid Friendly Appliance
*      Author  : Steffen Lang, PNNL
*
*      performs the fft on the frequency
*      signal
*
*************************************/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "gfa.h"

double window[NUM_OF_SAMPLES];
unsigned char bit_reverse[NUM_OF_SAMPLES];

/*******************************************
*      init_fft()
*      ---------------------------------------
*      initialisation for fourier transformation
*      bitreverse ordering and windowing
*
*      parameter :
*       none
*
*      return :
*       void
*
*******************************************/
void init_fft()
{
       unsigned char i=0;
       unsigned char bit;
       int y;
       unsigned char reverse;

       for(y=0;y<NUM_OF_SAMPLES;y++)
       {
              /* bit reverses order */
              reverse = 0x00;
              for(bit=1;bit;bit<<=1)
              {
                     reverse<<=1;
                     if(bit&i)
                            reverse|=0x01;
              }
              bit_reverse[i++] = reverse;

              /* create window function */
              window[y] = 1;             //default : no windowing

              #ifdef HANNING2                    //hanning window
                     window[y] = (1-cos(2*PI*y/NUM_OF_SAMPLES))/2;
              #endif
              #ifdef HANNING2SQRT        //hanning window sqrt
                     window[y] = (1-cos(2*PI*y/NUM_OF_SAMPLES))/2;
                     window[y] = window[y]*window[y];
              #endif

              #ifdef HANNING2SQRTHALF    //half hanning window sqrt
                     window[y] = (1-cos(PI*y/NUM_OF_SAMPLES))/2;
```

```
                    window[y] = window[y]*window[y];
                #endif
        }
}

/********************************************
*       power_fft()
*       ----------------------------------------
*       - performs the fft ( only magnitude ) on
*      the time signal
*       - the result is also saved in this array
*      ( overwritten )
*
*       parameter :
*        samples  - time signal, array of 256 values
*        spectrum - output for power spectrum
*
*       return :
*        always 0
********************************************/
int power_fft( double* samples, double* spectrum )
{
    unsigned i, j, k, n;
    unsigned mmax, m;
        double   temp;
    double   twoPI = 2.0 * PI;
    double   temp_real, temp_img;     /* temp real, temp imaginary */

        double   im_out[NUM_OF_SAMPLES];
        double   re_in[NUM_OF_SAMPLES];

        //copy samples into re_in and perfrom hanning window
        for ( i=0; i < NUM_OF_SAMPLES; i++ )
        {
                re_in[i]=samples[i]*window[i];
        }

        //order bitreverse
        for ( i=0; i < NUM_OF_SAMPLES; i++ )
        {
                if(i<bit_reverse[i])
                {
                        temp = re_in[i];
                        re_in[i] = re_in[bit_reverse[i]];
                        re_in[bit_reverse[i]] = temp;
                }
                im_out[i] = 0.0;
        }

        //perform fft
    m = 1;
    mmax = 2;
    while (mmax <= NUM_OF_SAMPLES)             //2,4,8,16,32,64,128
    {
         double theta = twoPI / (double)mmax;
         double sin_2theta = sin ( -2 * theta );
         double sin_theta = sin ( -theta );
         double cos_2theta = cos ( -2 * theta );
         double cos_theta = cos ( -theta );
         double w = 2 * cos_theta;
         double real_buf[3], img_buf[3];

         for ( i=0; i < NUM_OF_SAMPLES; i += mmax )
        {
            real_buf[1] = cos_theta;
                    real_buf[2] = cos_2theta;
            img_buf[1] = sin_theta;
            img_buf[2] = sin_2theta;

            for ( j=i, n=0; n < m; j++, n++ )
            {
                real_buf[0] = w*real_buf[1] - real_buf[2];
                real_buf[2] = real_buf[1];
                real_buf[1] = real_buf[0];

                img_buf[0] = w*img_buf[1] - img_buf[2];
                img_buf[2] = img_buf[1];
                img_buf[1] = img_buf[0];

                k = j + m;
                temp_real = real_buf[0]*re_in[k]  - img_buf[0]*im_out[k];
                temp_img  = real_buf[0]*im_out[k] + img_buf[0]*re_in[k];

                re_in[k]  = re_in[j]  - temp_real;
                im_out[k] = im_out[j] - temp_img;

                re_in[j] += temp_real;
                im_out[j]+= temp_img;
            }
        }
        m = mmax;
        mmax <<= 1;
    }

       //get the power spectrum
       for ( i=0; i < NUM_OF_SAMPLES; i++ )
       {
             re_in[i]  /=NUM_OF_SAMPLES;
        im_out[i] /=NUM_OF_SAMPLES;
             spectrum[i] = sqrt( re_in[i]*re_in[i] + im_out[i]*im_out[i] );
             #ifdef FFT_IN_DB
                    spectrum[i] = 20 * log10(spectrum[i]);
             #endif
       }

       return 0;
}
```

```
/*************************************
*      File    : gfa_if.c
*      Project : Grid Friendly Appliance
*      Author  : Steffen Lang, PNNL
*
*      interface channel to the data
*
*************************************/


/* include files */
#include "gfa.h"
#include <time.h>
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <signal.h>
#include <unistd.h>
#include <sys/time.h>
#include <fcntl.h>
#include <sys/stat.h> /* for mode definitions */

/* imported variable */
extern double samples [];
extern int  analysis_running;
extern int  new_data_avail;
extern int  end_of_file;
extern char log_time[];
extern int  operation_mode;
extern int  logfile_type;

/* static variables */
static FILE * fp;
static int (*get_samples_func)(void);
static int char_per_line;
static struct tm * tm_logstart;
static struct tm * tm_request;

char buf[4096];

/*prototypes*/
static int get_samples_realtime(void);
static int get_samples_logfile(void);
static int position_filepointer(char* argv2);
static int str2date1(char *s, struct tm* t);
static int str2date2(char *s, struct tm* t);
static int str2date3(char *s, struct tm* t);
static int month_convert(char *s);

/*******************************************
*      init_interface()
*      ---------------------------------------
*      parameter :
*       argc   - same as in command line
*       argv
*
*      return : 0  = OK
*                    1  = ERROR
*******************************************/
void init_interface(int argc, char *argv[])
```

```
{
       char freq_file[100];
       struct itimerval itimer;

       //find out the mode
       if(argc==1)
       {
              operation_mode = REALTIME;
              get_samples_func = get_samples_realtime;
              strcpy(freq_file,PROCFILE_DX2);
       }
       else
       {
              char *ptr;
              if(argc!=3 && argc!=4)
              {
                     printf("Illegal number of comand line parameter (%d)\n",argc);
                     help();
                     exit(0);
              }

              //check file extension
              if((ptr=strchr(argv[1],'.'))==0)
              {
                     printf("Error in parameter\n");
                     help();
                     exit(0);
              }

              if(strcmp(ptr,".conv")==0)
              {
                     logfile_type  = TYPE_CONV;
                     char_per_line = 29;
              }
              else if(strcmp(ptr,".fsu")==0)
              {
                     logfile_type  = TYPE_FSU;
                     char_per_line = 31;
              }
              else if(strcmp(ptr,".psm")==0)
              {
                     logfile_type  = TYPE_PSM;
              }
              else
              {
                     printf("Extension of <%s> is illegal\n",argv[1]);
                     help();
                     exit(0);
              }
              operation_mode = LOGFILE;
              get_samples_func = get_samples_logfile;
              strcpy(freq_file,argv[1]);
       }

       //open the file that contains the grid frequency
       //either the /proc file or the log file
       if((fp=fopen(freq_file,"r"))==NULL)
       {
              printf("ERROR : Can not open '%s'\n",freq_file);
              exit(0);
       }

       //position in logged data file
```

```
	if(operation_mode==LOGFILE)
	{
		if(position_filepointer(argv[2]))
		{
			exit(0);
		}
	}

	if(argc<4)			//no histogram - setup timer
	{
		//setup alarm
		itimer.it_value.tv_sec     = TIMER_INTERVAL;		//time	until
first occurance
		itimer.it_value.tv_usec    = 00;
		itimer.it_interval.tv_sec  = TIMER_INTERVAL;		//timer interval
		itimer.it_interval.tv_usec = 00;
		signal(SIGALRM,get_samples);
		setitimer(ITIMER_REAL, &itimer, NULL);
	}

	return;
}

/*******************************************
*	get_samples()
*   ----------------------------------------
*   This function is invoked by the
*	signal SIGALRM in certain time intervals.
*	( defined by TIMER_INTERVAL in gfa.h )
*
*	parameter :
*	 sig_val - type of alarm
*
*
*******************************************/
void get_samples(int sig_val)
{
	//gettimeofday(&start,(struct timezone *)0);
	if(sig_val!=SIGALRM)
		return;

	(*get_samples_func)();

	new_data_avail = 1;

	//gettimeofday(&end,(struct timezone *)0);
	//printf("Start sec: %d , usec: %d\n",start.tv_sec,start.tv_usec);
	//printf("End   sec: %d , usec: %d\n",end.tv_sec,end.tv_usec);
}

/*******************************************
*	get_samples_realtime()
*   ----------------------------------------
*	reads NUM_OF_SAMPLES samples from the /proc
*	file 54 and saves them to global array
*	'samples'
*
*	parameter :
*	 none
*
*	return :
*	 none
*******************************************/
```

```
static int get_samples_realtime(void)
{
       #define BYTES_PER_LINE     8
       #define LINES_IN_FILE    NUM_OF_SAMPLES
       #define BYTES_IN_FILE    (BYTES_PER_LINE*LINES_IN_FILE)

       int len;
       int i=0;
       char * ptr;
       char sfreq[10];
       double freq;

       if(analysis_running)
              printf("ERROR : Time problem. fft is running while read new data \n");

       rewind(fp);
       fflush(fp);                //reset file

       /* read all data */
       if((len=fread(buf,1,4096,fp))<=0)
       {
              printf("ERROR in fread from proc file. Can nor read\n");
              return 1;
       }

       if(len!=BYTES_IN_FILE)
       {
              printf("ERROR : too less data : only %d byte\n",len);
              return 1;
       }

       for(i=0;i<NUM_OF_SAMPLES;i++)
       {
              ptr = buf + (i*BYTES_PER_LINE);         //set pointer to start of next
value
              strncpy(sfreq,ptr,7);                          //format : '60.1234' = 7
characters
              sfreq[7]='\0';
              freq = atof(sfreq);
              if(freq<=0.0)
              {
                     printf("ERROR in format. string is %s freq is %f\n",sfreq,freq);
              }
              else
              {
                     samples[i] = freq;
                     //printf("f[%d]: %6.4f \n",i, data[i]);
              }
       }
       return 0;
}

//#define CHAR_PER_LINE 29

/*******************************************
*      get_samples_logfile()
*   ---------------------------------------
*      reads new samples from logfile and
*      rearranges the global array 'samples'
*
*      parameter :
*       none
*
```

```
*	return :
*	 none
*******************************************/
static int get_samples_logfile(void)
{
	int  i;
	char buf[100];
	char sfreq[10];
	char * ptr;
	static int first_time = 1;

	if(first_time)			//just read on line to make sure that
		fgets(buf,50,fp);	//filepointer points to beginning of a line

	for(i=0;i<NUM_OF_SAMPLES;i++)
    {
	if(i<NUM_OF_SAMPLES-(TIMER_INTERVAL*10) && !first_time)
		{
			//rearrange the array
			samples[i] = samples[i+TIMER_INTERVAL*10];
		}
		else
		{
			//get new samples from logfile
		if(fgets(buf,50,fp)==NULL)
		{
			end_of_file = 1;
			return 0;
		}

		//process data form file
		if(logfile_type==TYPE_FSU)
		{
			ptr = &buf[23];		//ptr points to the frequency (*.fsu
file)
			strncpy(sfreq,ptr,6);
				sfreq[6]='\0';
		}
		else if(logfile_type==TYPE_CONV)
		{
			ptr = &buf[21];		//ptr points to the frequency (*.conv
file)
			strncpy(sfreq,ptr,6);
				sfreq[6]='\0';
		}
		else if(logfile_type==TYPE_PSM)
		{
			if((ptr=strrchr(buf,'.'))==NULL)
			{
				printf("Error in format in logfile\n");
				exit(0);
			}
			ptr-=2;				//ptr points to the frequency
(*.psm file)
			strncpy(sfreq,ptr,7);
				sfreq[7]='\0';
		}

			if(i==NUM_OF_SAMPLES-1)		//if the last one
			{
				//update string that represents current time
				if(logfile_type==TYPE_FSU)
				{
```

```
                                strncpy(log_time,buf,20);
                                log_time[13] = ':';
                                log_time[16] = ':';
                        }
                        else if(logfile_type==TYPE_CONV)
                        {
                                strncpy(log_time,buf,18);
                                log_time[11] = ':';
                                log_time[14] = ':';
                        }
                        else if(logfile_type==TYPE_PSM)
                        {
                                char stime[10];
                                struct tm *tm_temp;
                                time_t time_temp;

                                sscanf(buf,"%s",stime);
                                time_temp = mktime(tm_logstart) + atof(stime);
                                tm_temp = localtime(&time_temp);
                                sprintf(log_time,"%s",asctime(tm_temp));
                                log_time[strlen(log_time)-1] = '\0';
                        }
                }
                samples[i] = atof(sfreq);
        }
   }
   first_time = 0;

        return 0;
}

/*********************************************
*       position_filepointer()
*       ----------------------------------------
*       opens the logfile, searchs for the start
*       line, set filepointer to start
*
*
*       return : 0  = OK
*                    1  = ERROR
*********************************************/
static int position_filepointer(char* argv2)
{
        char buf[100];

        double diff;
        time_t time_r, time_l;

        tm_logstart = malloc(sizeof(struct tm));
        tm_request  = malloc(sizeof(struct tm));

        if((logfile_type==TYPE_CONV) || (logfile_type==TYPE_CONV))
        {
                char search_date[30];
                unsigned long temp_line = 0;
                unsigned long offset;
                int date_len;

                // get date from command line parameter
                if(str2date1(argv2, tm_request))
                {
                        printf("Illegal date format : %s\n",argv2);
                        fclose(fp);
			return 1;
		}

		// get first date in log file
		if(fseek(fp,0,SEEK_SET))
		{
			printf("Error in logfile - fseek\n");
			fclose(fp);
			return 1;
		}
		if(logfile_type == TYPE_FSU)
		{
			fgets(buf,40,fp);		//skip first line
			fgets(buf,40,fp);
			buf[19]='\0';
			if(str2date2(buf, tm_logstart))
			{
				printf("Error  in  file  format  of  logfile.  <<%s>>  is
illegal1\n",buf);
				fclose(fp);
				return 1;
			}

			//printf("first   date   :   %d   %d   %d   ",tm_logstart.tm_mon,
tm_logstart.tm_mday, tm_logstart.tm_year );
		}
		else
		{
			fgets(buf,40,fp);
			buf[17] = '\0';
			if(str2date1(buf, tm_logstart))
			{
				printf("Error  in  file  format  of  logfile.  <<%s>>  is
illegal\n",buf);
				fclose(fp);
				return 1;
			}
		}

		//calculate approx. position
		time_r = mktime(tm_request);
		time_r-= (int)NUM_OF_SAMPLES/10;	//subtract time to position to
beginning
		free(tm_request);
		tm_request = localtime(&time_r);
		time_l = mktime(tm_logstart);
		diff = difftime(time_r,time_l);

		if(diff<1.00)
		{
			//printf("difference is : %f\n",diff);
			fclose(fp);
			printf("* An error causes the program to exit\n");
			printf("*  Date/time '%s' is not available in file1.\n",argv2);
			return 1;
		}
		temp_line = (unsigned long)(diff * 7.0);

		/* now start parsing */
		if(fseek(fp,temp_line*char_per_line,SEEK_SET))
		{
			fclose(fp);
			printf("* An error causes the program to exit\n");
```

```
                        printf("* Date/time '%s' is not available in file2.\n",argv2);
                        return 1;
                }

                if(logfile_type == TYPE_FSU)
                {
                        date_len = 19;
                        sprintf(search_date,"%d %02d %02d %02d %02d %02d",(tm_request-
>tm_year)+1900,(tm_request->tm_mon)+1,tm_request->tm_mday,tm_request-
>tm_hour,tm_request->tm_min,tm_request->tm_sec);
                }
                else
                {
                        date_len = 17;
                        sprintf(search_date,"%02d/%02d/%02d %02d %02d %02d",(tm_request-
>tm_mon)+1,        tm_request->tm_mday,        (tm_request->tm_year)-100,tm_request-
>tm_hour,tm_request->tm_min,tm_request->tm_sec);
                }

                do
                {
                        if(fgets(buf,30,fp)==NULL)        //read a line
                        {
                                fclose(fp);
                                printf("* An error causes the program to exit\n");
                                printf("*   Date/time   '%s'   is   not   available   in
file3.\n",argv2);
                                return 1;
                        }
                        //buf[date_len] = '\0';
                        //printf("%s",buf);
                        //getchar();
                }while(strncmp(buf,search_date,date_len));


                offset = ftell(fp) - char_per_line - 10;
                fseek(fp,offset,SEEK_SET);

                free(tm_logstart);
                return 0;
        }
        else if(logfile_type==TYPE_PSM)
        {
                char * ret;
                char temp[60];
                double search;
                int u;

                // get date from command line parameter -> store in tm_request
                if(str2date1(argv2, tm_request))
                {
                        printf("Illegal date format : %s\n",argv2);
                        fclose(fp);
                        return 1;
                }

                // get first date in log file  -> store in tm_logstart
                while((ret=fgets(buf,60,fp))!=NULL)
                {
                        #define FIND_IT      "reference time"
                        if(strncmp(buf,FIND_IT,strlen(FIND_IT))==0)
                        {
                                break;
                }
            }
            if(!ret)
            {
                printf("file   format   is   incorrect   -   no   string   '%s'
found\n",FIND_IT);
                exit(0);
            }
            fgets(buf,60,fp);          //read reference time
            if(str2date3(buf, tm_logstart))
            {
                printf("Error   in   file   format   of   logfile.   <<%s>>   is
illegal1\n",buf);
                fclose(fp);
                return 1;
            }

            //get difference
            time_r = mktime(tm_request);
            time_r-= (int)NUM_OF_SAMPLES/10;        //subtract  time  (25  sec)  to
position to beginning
            free(tm_request);
            tm_request = localtime(&time_r);
            time_l = mktime(tm_logstart);
            diff = difftime(time_r,time_l);

            if(diff<1.00)
            {
                fclose(fp);
                printf("* An error causes the program to exit\n");
                printf("*  Date/time '%s' is not available in file1.\n",argv2);
                return 1;
            }

            for(u=0;u<10;u++)   fgets(buf,60,fp);   //skip some lines


            //position the file pointer - sequential search
            do
            {
                fgets(buf,60,fp);          //read line per line
                sscanf(buf,"%s",temp);
                search = atof(temp);

            }while(search<diff);

            return 0;
        }
        else
        {
            //logfile_type not selected
            return 1;
        }
}

/*********************************************
*       str2date1() - format in *.conv files
*                       anf command line
*       ---------------------------------------
*       conversion of string into struct tm
*
*       parameter :
*     s  - char array with the date string
*               format is 'mm/dd/yy/hh/mm/ss'
*    t  - struct tm; represents time and date
*
*      return :
*       0  - OK
*       1  - ERROR
********************************************/
static int str2date1(char *s, struct tm* t )
{
       if(strlen(s)!=17)
              return 1;

       t->tm_mon=10*(s[0]-0x30);
       t->tm_mon+=s[1]-0x30;
       t->tm_mday=10*(s[3]-0x30);
       t->tm_mday+=s[4]-0x30;
       t->tm_year=10*(s[6]-0x30);
       t->tm_year+=s[7]-0x30;
       t->tm_hour=10*(s[9]-0x30);
       t->tm_hour+=s[10]-0x30;
       t->tm_min=10*(s[12]-0x30);
       t->tm_min+=s[13]-0x30;
       t->tm_sec=10*(s[15]-0x30);
       t->tm_sec+=s[16]-0x30;
       t->tm_isdst = 0;

       /* check if everything is in the range */
       if(    (  (t->tm_mon)<1 ||  (t->tm_mon)>12 ) ||
              ( (t->tm_mday)<1 || (t->tm_mday)>31 ) ||
              ( (t->tm_year)>5 && (t->tm_year)<95 ) ||
              ( (t->tm_year)<0                    ) ||
              ( (t->tm_hour)<0 || (t->tm_hour)>24 ) ||
              (  (t->tm_min)<0 ||  (t->tm_min)>59 )  )
       {
              return 1;
       }

       //convert to tm format
       t->tm_mon-=1;       /* month : 0 - 11 */
       t->tm_hour-=1;             /* hour  : 0 - 23 */
       if(t->tm_year > 0 && t->tm_year <=20 )
              t->tm_year+=100;

       return 0;
}

/********************************************
*      str2date2() - format in *.fsu files
*      ----------------------------------------
*      conversion of string into struct tm
*
*      parameter :
*    s  - char array with the date string
*               format is yyyy/mm/dd/hh/mm
*    t  - struct tm; represents time and date
*
*      return :
*       0  - OK
*       1  - ERROR
********************************************/
static int str2date2(char *s, struct tm* t )
{
       if(strlen(s)!=19)
```

```
        {
                //printf("too less");
                return 1;
        }

        t->tm_mon=10*(s[5]-0x30);
        t->tm_mon+=s[6]-0x30;

        t->tm_mday=10*(s[8]-0x30);
        t->tm_mday+=s[9]-0x30;

        t->tm_year=1000*(s[0]-0x30);
        t->tm_year+=100*(s[1]-0x30);
        t->tm_year+=10*(s[2]-0x30);
        t->tm_year+=s[3]-0x30;

        t->tm_hour=10*(s[11]-0x30);
        t->tm_hour+=s[12]-0x30;

        t->tm_min=10*(s[14]-0x30);
        t->tm_min+=s[15]-0x30;

        t->tm_sec   = 0;

        t->tm_isdst = 0;


        //printf("%d %d %d %d %d\n",t->tm_mon, t->tm_mday, t->tm_year, t->tm_hour, t-
>tm_min );
        /* check if everything is in the range */
        if(    (  (t->tm_mon)<1 ||  (t->tm_mon)>12 ) ||
                ( (t->tm_mday)<1 || (t->tm_mday)>31 ) ||
                ( (t->tm_year)>2005 && (t->tm_year)<1995 ) ||
                ( (t->tm_hour)<0 || (t->tm_hour)>24 ) ||
                (  (t->tm_min)<0 ||  (t->tm_min)>59 )  )
        {
                return 1;
        }

        //convert to tm format
        t->tm_mon-=1;       /* month : 0 - 11   */
        t->tm_hour-=1;              /* hour  : 0 - 23 */
        t->tm_year-=1900;   /* years since 1900 */

        return 0;
}

/********************************************
*       str2date3() - format in *.psm files
*       ----------------------------------------
*       conversion of string into struct tm
*
*       parameter :
*     s  - char array with the reference time
*        format example : '10-Aug-1996 22:35:30.000'
*
*     t  - struct tm; represents time and date
*
*       return :
*        0  - OK
*        1  - ERROR
********************************************/
static int str2date3(char *s, struct tm* t )
```

```
{
       char month[4];

       strncpy(month,s+3,3);
       month[3] = '\0';

       if(month_convert(month))
       {
              return 1;
       }

       t->tm_mon=10*(month[0]-0x30);
       t->tm_mon+=month[1]-0x30;

       t->tm_mday=10*(s[0]-0x30);
       t->tm_mday+=s[1]-0x30;

       t->tm_year=1000*(s[7]-0x30);
       t->tm_year+=100*(s[8]-0x30);
       t->tm_year+=10*(s[9]-0x30);
       t->tm_year+=s[10]-0x30;

       t->tm_hour=10*(s[12]-0x30);
       t->tm_hour+=s[13]-0x30;

       t->tm_min=10*(s[15]-0x30);
       t->tm_min+=s[16]-0x30;

       t->tm_sec=10*(s[18]-0x30);
       t->tm_sec+=s[19]-0x30;

       t->tm_isdst = 0;

       if(    (  (t->tm_mon)<1 ||  (t->tm_mon)>12 ) ||
              ( (t->tm_mday)<1 || (t->tm_mday)>31 ) ||
              ( (t->tm_year)>2005 && (t->tm_year)<1995 ) ||
              ( (t->tm_hour)<0 || (t->tm_hour)>24 ) ||
              (  (t->tm_min)<0 ||  (t->tm_min)>59 )  )
       {
              return 1;
       }

       //convert to tm format
       t->tm_mon-=1;       /* month : 0 - 11   */
       t->tm_hour-=1;             /* hour  : 0 - 23 */
       t->tm_year-=1900;   /* years since 1900 */

       return 0;
}

/*********************************************
*      month_convert()
*      -----------------------------------------
*      overwrittes the given string ( e.g "jan") by the
*      corresponding number ("01")
*
*      parameter :
*    s  - char array that contains the month
*              it will be overwritten by the number
*              of that month
*
*      return :
*       0  - OK
*       1  - ERROR
*******************************************/
static int month_convert(char *s)
{
       int i=0;
       char                              *month[]                              =
{"JAN","FEB","MAR","APR","MAY","JUN","JUL","AUG","SEP","OCT","NOV","DEC"};

       s[0] = toupper((int)s[0]);
       s[1] = toupper((int)s[1]);
       s[2] = toupper((int)s[2]);

       while(i<12)
       {
              if(strcmp(s,month[i])==0)
              {
                     break;
              }
              i++;
       }

       if(i>=12)
       {
              return 1;
       }
       else
       {
              sprintf(s,"%02d",i+1);
       }
       return 0;
}
```

```c
/*************************************
*      File    : gfa_server.c
*      Project : Grid Friendly Appliance
*      Author  : Steffen Lang, PNNL
*
*      tcp/ip handling
*
*************************************/

/* include files */
#include "gfa.h"
#include<sys/socket.h>
#include<sys/types.h>
#include<arpa/inet.h>
#include<unistd.h>
#include<fcntl.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

/* extern variables */
extern double spectrum[NUM_OF_SAMPLES];
extern double samples [NUM_OF_SAMPLES];
extern PEAK   peaks[NUM_OF_SAMPLES/2];
extern double threshold[5];
extern double thres_integral[5];
extern double stddevfft;
extern server_state state;
extern char log_time[];
extern int  operation_mode;

/* global variables */
static int sock_desc_server;
static int sock_desc_client ;
static struct sockaddr_in server;
static struct sockaddr_in client;
static int first_package = 0;

/* prototypes */
static int server_setup();
static int server_get_transmission_string(char *s);

/*******************************************
*      server_handler()
*      ----------------------------------------
*      handles the server functionality
*
*      parameter :
*       none
*
*      return :
*       void
*
*******************************************/
void server_handler()
{
       //printf("sm *%d*\n",state);
       if(state == NO_SOCKET )
       {
             if(!server_setup())
             {
                    state = SOCKET;
             }
```

```
        }
        else if(state==SOCKET)
        {
                int client_size;

                client_size = sizeof(client);
                sock_desc_client = accept(sock_desc_server, &client, &client_size);
                if(sock_desc_client==-1)
                {
                        return;
                }
                else
                {
                        state = CLIENT;
                        fcntl(sock_desc_client,F_SETFL,O_NONBLOCK);    //set client non-
blocking
                        printf("client from %s port : %d\n", inet_ntoa(client.sin_addr),
client.sin_port);
                }
        }
        else if(state==CLIENT || state==REQUEST)
        {
                int bytes;
                char in[1024];
                char out[4096] = "";

                bytes = recv(sock_desc_client, in, 256, 0);
             if (bytes <= 0)
             {
                //printf("not read\n");
             }
             else
             {
                in[bytes] = '\0';
                //printf("Client sent <%s>\n", in);
                    if(strcmp(in,"request")==0)
                    {
                       state = REQUEST;
                       first_package = 1;
                    }
                    else if(strcmp(in,"stop")==0)
                    {
                       state = CLIENT;
                       return;
                    }
                    else if(strcmp(in,"close")==0)
                    {
                       close(sock_desc_client);
                               printf("socket closed\n");
                               state = SOCKET;
                               return;
                    }
                }

                if(state==REQUEST)
                {
                        server_get_transmission_string(out);
                        first_package = 0;
                        if(send(sock_desc_client,out,strlen(out),0)==-1)
                    {
                       printf("Can not send\n");
                       close(sock_desc_client);
                       state = SOCKET;
                    }
                    else
                    {
                       //printf("send OK   \n");
                    }
                }
        }
}

/*********************************************
*       server_get_transmission_string()
*       ----------------------------------------
*       assembles the string that will be
*       transmitted to the client
*
*       parameter :
*        s  - string where information is stored
*
*       return :
*        always 0
*
*********************************************/
static int server_get_transmission_string(char *s)
{
        char temp[150];
        int i;
        int first_sample = 246;          // we only transmit 10 samples of the time
signal

        //spectrum *************
        strcpy(s,"<spectrum>");
        for(i=0;i<52;i++)
        {
                sprintf(temp,"%5.3f",spectrum[i]);
                if(i<51)
                {
                        strcat(temp,",");
                }
                strcat(s,temp);
        }
        strcat(s,"</spectrum>");

        //samples  *************
        strcat(s," <samples>");
        if(first_package)          //in first package : transmit 11 samples ( 0 - 10
)
                first_sample = 245;

        for(i=first_sample;i<NUM_OF_SAMPLES;i++)
        {
                sprintf(temp,"%5.3f",(samples[i]+samples[i-1])/2);
                if(i!=NUM_OF_SAMPLES-1)
                {
                        strcat(temp,",");
                }
                strcat(s,temp);
        }
        strcat(s," </samples>");

        //peaks   *************
        strcat(s," <peaks>");
        i=0;
        while(peaks[i].pos>=0)
```

```
	{
		sprintf(temp,"%d",peaks[i].pos);
		if(peaks[i+1].pos>=0)
		{
			strcat(temp,",");
		}
		strcat(s,temp);
		i++;
	}
	strcat(s," </peaks>");

	//details   *************
	strcat(s," <details>");
	strcat(s,"<peak>");
	i=0;
	while(peaks[i].pos>=0)
	{
		sprintf(temp,"%5.3f      Hz            %5.3f      dB            %5.3f°
%5.3f\n",peaks[i].pos*0.0039,
				peaks[i].value,peaks[i].angle,peaks[i].sharp);
		strcat(s,temp);
		i++;
	}
	strcat(s,"</peak> <integral>");
	for(i=0;i<5;i++)
	{
		sprintf(temp,"%5.3f %5.3f\n",threshold[i],thres_integral[i]);
		strcat(s,temp);
	}
	strcat(s,"</integral><stddev>");
	sprintf(temp,"%6.3f",stddevfft);
	strcat(s,temp);
	strcat(s,"</stddev>");
	strcat(s," </details>");

	//time
	strcat(s," <time>");
	if(operation_mode==LOGFILE)
	{
		strcat(s,log_time);
	}
	else
	{
		strcat(s,"   ");
	}
	strcat(s," </time>");

	return 0;
}

/*******************************************
*	server_setup()
*	-----------------------------------------
*	creating socket and listen to it
*
*	parameter :
*	 none
*
*	return :
*	 0 = OK
*	 1 = ERROR
*
*******************************************/
```

```
static int server_setup()
{
        //create socket
        if((sock_desc_server=socket(AF_INET,SOCK_STREAM,0))==-1)
        {
                printf("Error in socket() !\n");
        return 1;
        }
        server.sin_family=AF_INET;
    server.sin_addr.s_addr=INADDR_ANY;
    server.sin_port=htons(3552); //we are using

    //set server as non-blocking
    fcntl(sock_desc_server,F_SETFL,O_NONBLOCK);

        //bind
    if(bind(sock_desc_server,(struct sockaddr *)&server,sizeof(server))==-1)
    {
        printf("Error in bind() !\n");
        return 1;
        }

        //listen
        if(listen(sock_desc_server,1)==-1)
        {
        printf("Error in listen() !\n");
        return 1;
        }
        return 0;
}
```