

**A MULTIPROCESSING PLATFORM FOR
TRANSIENT EVENT DETECTION**

by

UMAIR A. KHAN

**S.B. MATHEMATICS WITH COMPUTER SCIENCE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
(MAY 1992)**

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

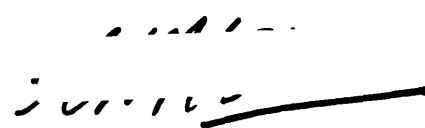
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

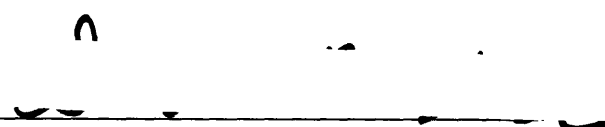
© Umair A. Khan 1995
All rights reserved

The author hereby grants to MIT permission to reproduce and to
distribute publicly paper and electronic copies of this thesis
document in whole or in part.

Signature of Author _____


Department of Electrical Engineering and Computer Science
May 26, 1995

Certified by _____


Steven B. Leeb
Carl Richard Soderberg Assistant Professor of Power Engineering
Thesis Supervisor

Accepted by _____

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY


F.R. Morgenthaler
Chairman, Department Committee on Graduate Theses

MAR 18 1996

Barker Eng

A Multiprocessing Platform for Transient Event Detection

by

Umair A. Khan

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

Abstract

Conventional nonintrusive load monitoring relies on measurements of steady-state current and voltage for determining the operating schedule of loads of interest in a building. The Multiscale Transient Event Detector (TED) described in [1] advances the capabilities of conventional NILMs by using vector space methods to identify load transients or transient sections. The structure of the TED algorithm has significant parallelism inherent to it. This suggests the possibility of implementing the TED as a multiprocessing machine with several inexpensive processors performing the various tasks in parallel. The development of a parallel version of the sequential algorithm, the synthesis of a model for a multiprocessing platform for transient event detection based on the parallel algorithm, and the design and construction of a prototype Multiprocessing Load Monitor (MLM), were the major goals of this thesis. Data dependencies in the TED algorithm were identified and used to parallelize the TED algorithm. A model for the MLM was then developed keeping in mind complexity and reliability issues. Hardware design of the main subsystems was undertaken. The MLM prototype consists of a *data acquisition front-end* and several *computational units* operating in parallel and communicating with one another. The acquisition front-end samples the analog input streams (envelopes of real and reactive power, etc.), stores and periodically transfers blocks of the digitized data to the computational modules. These 80C196-based modules may be configured to perform transient search on various time scales, time scaling of data using tree-structured decomposition [1], or result collation. They were designed to be scalable, and versatile enough to perform any of the TED operations. The MLM's user interface is implemented on a host PC, and allows, for instance, the downloading of code to the slave processors and the uploading of event detection results. The principal application of the prototype MLM was transient event detection on power load lines. The model, as well as the prototype, are general enough to allow transient detection in other environments.

Thesis Supervisor: Steven B. Leeb

Title: Carl Richard Soderberg Assistant Professor of Power Engineering

*How could a blade of grass
Repay the warmth from the Spring Sun.*

— Meng Chio [751–814]

To my mother and my father

Contents

1	Introduction and Background	13
1.1	Theoretical Background	13
1.2	Contributions of This Work	17
1.3	Thesis Outline	19
2	The Structure of the MLM	21
2.1	Parallelism in the TED Algorithm	21
2.1.1	Data Dependencies for TED Operations	22
2.1.2	The Parallel Transient Event Detection Algorithm	24
2.2	Complexity/Reliability Tradeoffs	25
2.3	The Structure of the MLM	32
3	The Data Acquisition Front-End	37
3.1	The Analog Preprocessor	37
3.2	Functional Overview of the Master Board	39
3.2.1	Data Acquisition and Storage	39
3.2.2	Data Transfer to Slave Processors	40
3.2.3	Host PC Communication	41
3.2.4	Control Logic	42
3.3	Design and Implementation	42
3.3.1	A/D Conversion	42
3.3.2	Data Storage	43
3.3.3	Slave Interface	46
3.3.4	Control Logic	47
3.3.5	Sampling Rate Generator	50

3.3.6	Reset Circuitry	51
3.3.7	PC Interface	51
3.3.8	Miscellaneous Hardware Components	53
3.4	Hardware Specifications	54
4	The MLM Computational Units	56
4.1	Design Overview of a Slave Module	56
4.1.1	The Processing Engine	57
4.1.2	PC Interface	57
4.1.3	Master Board Interface	60
4.1.4	Inter-Slave Communication	60
4.2	Functional Overview of the Slave Module	61
4.2.1	Event Detection	62
4.2.2	Tree-Structured Decomposition	66
4.2.3	Result Collation	66
4.3	Slave Module Implementation Details	67
4.3.1	Memory Section	67
4.3.2	Microprocessing Unit and Microcontrol PAL	68
4.3.3	PC Interface	69
4.3.4	Interconnection Details	70
4.3.5	Reset Circuitry	72
4.4	The MLM Slave Board	72
4.4.1	PC Interface Glue Logic	72
4.4.2	Jumper Selections	73
4.4.3	Component Layout	75
4.4.4	Hardware Specifications	76
5	Software Implementation of the MLM	77
5.1	Software for Slave Processors	77
5.1.1	Acquisition of Input Data	77
5.1.2	Pattern Search	78
5.1.3	Software Implementation of Load Chains	81
5.1.4	Template Management	82

5.1.5	Tree-structured Decomposition	83
5.1.6	Result Collation	85
5.1.7	PC Communication	87
5.2	Software Design of the PC interface	88
5.2.1	RISM Structure	90
5.2.2	ECM Program Structure	92
5.2.3	Communicating with the Master Board	93
5.2.4	Master Board Mode Selection	95
5.2.5	Selecting a Slave Processor for Communication	97
5.2.6	Program Code Output Format	97
5.2.7	Template Collection	98
5.2.8	Communicating with the Slave Processors	103
5.2.9	Collater Communication	106
5.2.10	Automatic Initialization of Slave Processors	109
6	Prototype Construction and Results	112
6.1	Development Cycle for the MLM	112
6.2	Prototype I : MLM-8S	113
6.3	Prototype II : MLM-16S	115
6.4	Results	118
6.5	Summary	132
7	Conclusion	133
	Bibliography	138
A	Master Board Schematics and Layouts	140
B	Slave Board Schematics and Layouts	148
C	PAL Code	156
C.1	Master Board PALs	156
C.2	Slave Board PALs	177
D	Software for Slave Processors	186

D.1	Code for V-section Search	187
D.2	Code for Tree-structured Decomposition	209
D.3	Code for Result Collation	232
D.4	Batch File for Code Compilation	240
E	Software for the Host PC Interface	241
E.1	ECM Code	241
E.2	Modifications Made to 80C196KC RISM	280
F	PC I/O Card for MLM Control	283
G	The Analog Preprocessor	286
H	Miscellaneous Details of Prototype Testing	294
H.1	V-section Sets	294
H.2	Software for Multiple Load Activation	295
I	The Genesis of the MLM	297

List of Figures

1.1	Rapid Start Lamp Bank Transients in (a) Real Power (b) Reactive Power	15
1.2	The Multiscale Transient Event Detection Algorithm	16
1.3	The Multiprocessing Load Monitor (MLM)	18
2.1	Data-Dependency Graph for TED Operations	24
2.2	The Parallel TED Algorithm	25
2.3	Improving Reliability By V-section Search Over More Streams	30
2.4	Identifying a Load on One Time Scale	32
2.5	Identifying N Loads on One Time Scale	33
2.6	A More Compact Load Assignment	34
2.7	The MLM Model	35
3.1	The MLM Master Board	40
4.1	The MLM Slave Module	58
4.2	The Interconnection Circuitry for a Slave Module	59
4.3	The MLM Model	62
4.4	Load Chains in the MLM	65
4.5	Slave Board Connectors	71
5.1	ECM Main Menu	93
5.2	Displaying Channel Data	95
5.3	Master Acquisition Mode Submenu	96
5.4	Slave Mode Selection Submenu	99
5.5	Slave Communication Submenu	101
5.6	Template Management Submenu	102

5.7	Special Function Register Submenu	105
6.1	Prototype Test Facility	118
6.2	MLMscope Report: Small Motor	121
6.3	MLMscope Report: Incandescent Light Bulbs	121
6.4	MLMscope Report: Instant Start Lamps	122
6.5	MLMscope Report: Rapid Start Lamps	122
6.6	MLMscope Report: Computer	123
6.7	MLMscope Report: Big Motor	123
6.8	MLMscope Report: Small Motor, Instant	126
6.9	MLMscope Report: Small Motor, Instant	126
6.10	MLMscope Report: Small Motor, Light	127
6.11	MLMscope Report: Rapid, Small Motor	127
6.12	MLMscope Report: Rapid, Instant	128
6.13	MLMscope Report: Rapid, Small Motor, Instant	128
6.14	MLMscope Report: Rapid, Instant, Small Motor	129
6.15	MLMscope Report: Small Motor, Light, Instant	129
6.16	MLMscope Report: Computer, Small Motor, Instant	130
6.17	MLMscope Report: Big Motor, Small Motor	130
6.18	MLMscope Report: Big Motor, Rapid, Small Motor, Instant	131
6.19	MLMscope Report: Big Motor, Small Motor, Rapid, Instant	131
A.1	Master Board Schematic 1	141
A.2	Master Board Schematic 2	142
A.3	Master Board Schematic 3	143
A.4	Master Board Schematic 4	144
A.5	Master Board Schematic 5	145
A.6	Master Board: PCB Component Layout	146
A.7	Master Board: PCB Component and Solder Sides	147
B.1	Slave Board Schematic 1	149
B.2	Slave Board Schematic 2	150
B.3	Slave Board Schematic 3	151

B.4	Slave Board Schematic 4	152
B.5	Slave Board Schematic 5	153
B.6	Slave Board: PCB Component Layout	154
B.7	Slave Board: PCB Component and Solder Sides	155
F.1	PC I/O Board Schematics	285
G.1	Signal Flow Graph	290
G.2	Envelope Preprocessor Block Diagram	291
G.3	Personal Computer Turn-On Transient	292

List of Tables

- 3.1 Master Board Component Listing 54
- 4.1 Slave Board Component Listing 76
- 5.1 Memory Map of the Slave Module 80C196KC 91
- 5.2 Naming Scheme for Raw V-section Template Files 99
- 6.1 Slave Module Configuration in the MLM-8S 113
- 6.2 Slave-Slave Interconnections for MLM-8S 114
- 6.3 Slave Module Configuration in the MLM-16S 115
- 6.4 Slave-Slave Interconnections for MLM-16S 116
- 6.5 Multiple Load Turn-on Delays 125

Acknowledgements

I am grateful to ESEERCo and the Electric Power Research Institute for funding this project. Essential hardware for this project was made available through a generous grant from the Intel Corporation. Test equipment used in the course of this thesis was made available through a generous grant from Tektronix.

I am deeply grateful to Steven Leeb, my thesis advisor, on-the-spot trainer, motivater, trouble shooter, friend and mentor. Thank you for the opportunity, the experience and for making sure I finished by May 1995.

I must thank my friends and colleagues: Ahmed Mitwalli for being the funniest man on earth and the most helpful; Steve Shaw for all his help and advice, and for being Steve Shaw; Rob Lepard, Mark Lee and Krishna Pandey for their significant contributions in building the MLM; Aaron, Deron, Neils, Kamakshi, Jeff Chapman, Ganish, Khurram, and Dave Otten for their advice and support. A special thanks to Vivian for all her cooperation throughout my stay at LEES.

I am not sure how to thank someone for bearing the burden of the worst of times and for being responsible for all my happiest moments. But I thank my wife, Zareen, anyway. Thanks too, to our daughter Samar who ensured by her arrival that I would finish up at MIT this year.

I thank my sister, Shazaf, for all the cards, letters, impersonations and private jokes. I am grateful to my father for supporting me in every decision and standing by me in every effort. My love for mathematics and for education, I owe to him.

Finally, I thank my mother. She cannot read these words, but then, she does not need to, as every word I have ever written, she wrote for me.

Chapter 1

Introduction and Background

1.1 Theoretical Background

The **Nonintrusive Load Monitor (NILM)** is a device that monitors the electric utility service entry of a building and, from measurements of voltage and aggregate current made solely at this point, is able to determine the operating schedule of every load of interest in the building. Nonintrusive load monitoring is a convenient and economical means of acquiring energy data for this purpose. Compared with conventional load monitoring, nonintrusive monitoring boasts easier installation, simplified data collection, and simplified data analysis because the NILM allows for all analysis to be done at a single central location [1]. Additionally, a NILM is a potentially powerful platform for power quality monitoring. “Power pollutants” are loads which do not draw sinusoidal input currents. Harmonic currents create harmonic voltages in the transmission system and degrade the quality of the delivered voltage waveform [2]. In conjunction with determining the operating schedule of the loads, the NILM could check for power quality offenders by relating the introduction of undesirable harmonics with the simultaneous turning on of the offending loads.

While the NILM demonstrates much improvement over conventional metering procedures in both ease and scope of operation, more sophisticated techniques are needed to deal with loads in an industrial or commercial setting. State-of-the-art nonintrusive monitoring devices such as the residential NILM [3], depend upon changes in steady state real and reactive power for load identification. Such a NILM would not for instance, be able to distinguish loads that coincidentally draw nearly identical steady state power. This presents a potential limitation in the context of industrial settings which contain loads that may be

modified to homogenize their steady state behavior [4].

The **Multiscale Transient Event Detector (TED)** described in [1] advances the capabilities of conventional NILMs by using vector space methods to identify load transients or transient sections. Instead of monitoring steady state conditions, it searches for transients and matches them to stored templates of the load whose activity caused them. Hence, it does not suffer from the limitations of a residential NILM in a commercial or industrial environment.

The basis for the TED is the observation that transient behavior of most important load classes is distinctive and repeatable. This allows reliable recognition of individual loads from the observed transients. Each transient in current, real or reactive power, and other quantities, may be subdivided into segments of *significant variation*, known as *v-sections*. A load may thus be modelled as a set of v-sections comprising its characteristic transient behavior. Detecting the turning on of a load would simply (and reliably) mean recognizing all the v-sections for the load in the input data stream. This pattern recognition may be done using any feasible pattern discrimination technique such as a euclidean filter or a transversal filter [5].

Figure 1.1, reproduced from [1], shows an example of a turn-on transient. Figure 1.1(a) shows the envelope of real power during the turn-on transient of a rapid start fluorescent lamp. The trace in Figure 1.1(b) shows the envelope of reactive power during the turn-on transient. The location of the v-sections in the two waveforms, computed by a change of mean detector implemented in MATLAB in [1], are schematically marked by ellipses A–E in the figures. The regions between the ellipses have very little variation in their level, i.e., are quasistatic. If two loads were to turn on simultaneously, the transients would overlap. This overlap, however, would *not* be intractable as long as the v-sections of one load transient occur during the quasistatic segments of the other load transient. As the widths of the v-sections are narrow, this may be quite probable, so that in most cases of overlap the TED would be able to resolve the loads correctly. By considering transients as sets of v-sections rather than as a single event to be detected, the TED allows for monitoring in a busy environment with a high rate of event generation.

The algorithm for the prototype Transient Event Detector (TED) of [1] is flowcharted in Figure 1.2 (reproduced from [1]).

The first step is data acquisition. Once it is armed, the TED waits for a triggering event

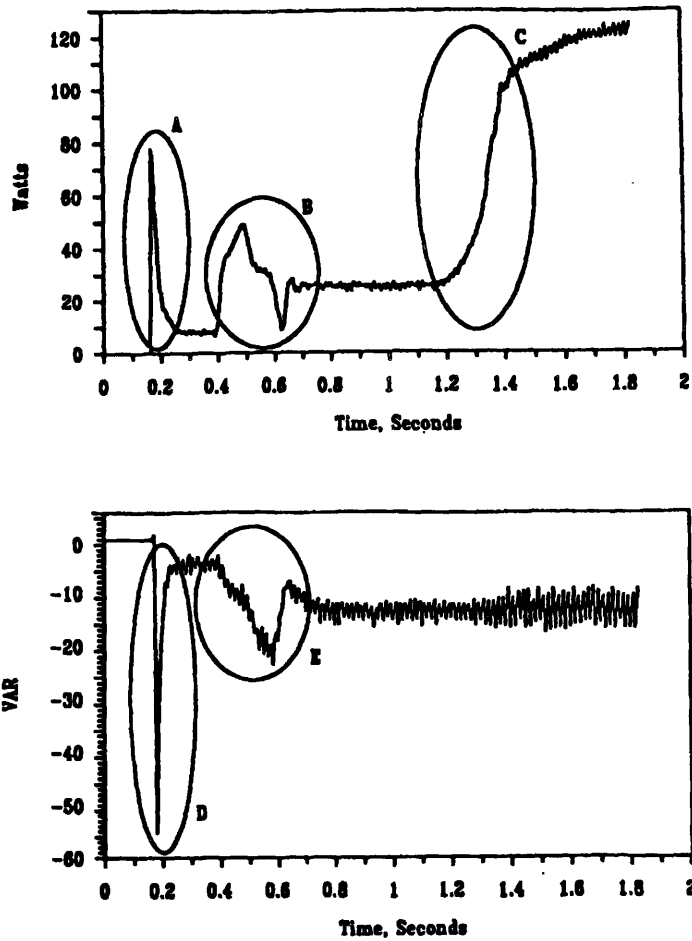


Figure 1.1: Rapid Start Lamp Bank Transients in (a) Real Power (b) Reactive Power

(a change of mean in the input data stream), whereupon a block of samples is acquired. The v-sections of the load transients will have been previously collected and stored as templates. (The discriminating filter searches for these templates in the input data in step 5.)

Event detection may be carried out over several time scales. For instance, the input stream may be downsampled by factors of two and analyzed for v-sections on those time scales. The advantage of such an approach would be to allow templates of complex v-sections to be reduced to manageable sizes which would conserve both memory and processing time. Hence, for instance, data stream decomposition would allow us to identify a particular transient on a coarser time scale than the input stream. The procedure employed by the TED for downsampling data is known as tree-structured decomposition and is performed

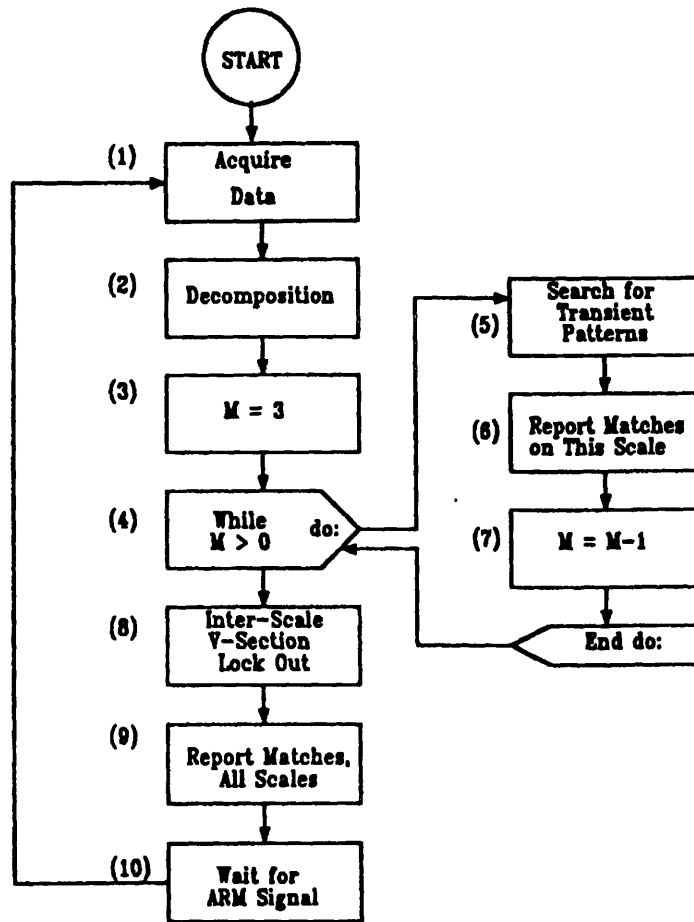


Figure 1.2: The Multiscale Transient Event Detection Algorithm

at step 2. In the prototype, the input data block is downsampled twice to produce data on three time scales. Steps 3 and 4 then set up a loop to perform pattern search on each scale. Transversal filtering is used to detect v-sections in the input data stream.

A major concern in searching for v-sections to establish load activity is the possibility of superimposed transients due to simultaneous activity of multiple loads. As noted above, the v-sections are small enough to make complete superpositioning unlikely, thus allowing some degree of overlap in the transients. In the rare event of complete superpositioning, pattern recognition could fail. The danger still remains however, of a v-section being falsely detected due to the activity of another load with a more complicated set of v-sections. If all the v-sections in a complex v-section set of a certain load are found, it is very likely that the transient associated with the load is present in the input stream. If now a v-

section belonging to a load with a simpler v-section set is also detected at about the same point in the input stream, it may be safely assumed that this v-section is not present and the identification is spurious. For instance, consider load X with a complicated transient consisting of four v-sections in an input stream, and load Y with only one v-section in the same input stream. If the v-section for Y resembles one of the v-sections for X, load Y may be falsely reported as activated each time load X turns on. To handle this case, an *intra-scale v-section lock out* is performed in step 5 on the result of the pattern search on each scale as follows: The pattern search on a scale is hierarchical, with v-sections belonging to the most complex transients being searched first. If a complex transient is detected, the locations of its v-sections are noted. A simpler transient will not be considered as identified if its v-sections are detected at the previously recorded, *locked out* locations.

Once all v-sections for a load on a time scale are found, the load transient is considered positively identified and a report is made of all events detected on that time scale (step 6). The TED then repeats event detection and lock out on the other time scales (step 7).

When all the patterns on all the scales have been searched, all events detected must be further conditioned by *inter-scale v-section lock out* (step 8). This follows the same principle as an intra-scale lockout. It guarantees that v-sections from a complex pattern on a coarse time scale were not used to match simpler v-sections on a finer time scale. Finally, a report of all loads identified is made in step 9, and the TED waits for the user to arm it again.

This multiscale transient event detection algorithm allows for several interesting routes for advancement. The event detection, tree-structured decomposition, and result collation procedures all have a high degree of parallelism inherent to them. This suggests the possibility of implementing the TED as a multiprocessing machine with several inexpensive processors performing the various tasks in parallel.

1.2 Contributions of This Work

The major contribution of this thesis was the development of a multiprocessing platform for multiscale transient event detection by extending the serial algorithm of [1]. The first step towards this goal was to analyze the TED algorithm in search for parallelism. The time/data dependencies of the major functional steps of this algorithm were brought out. Mapping these functions over a mesh of processors meant considering implementation issues early

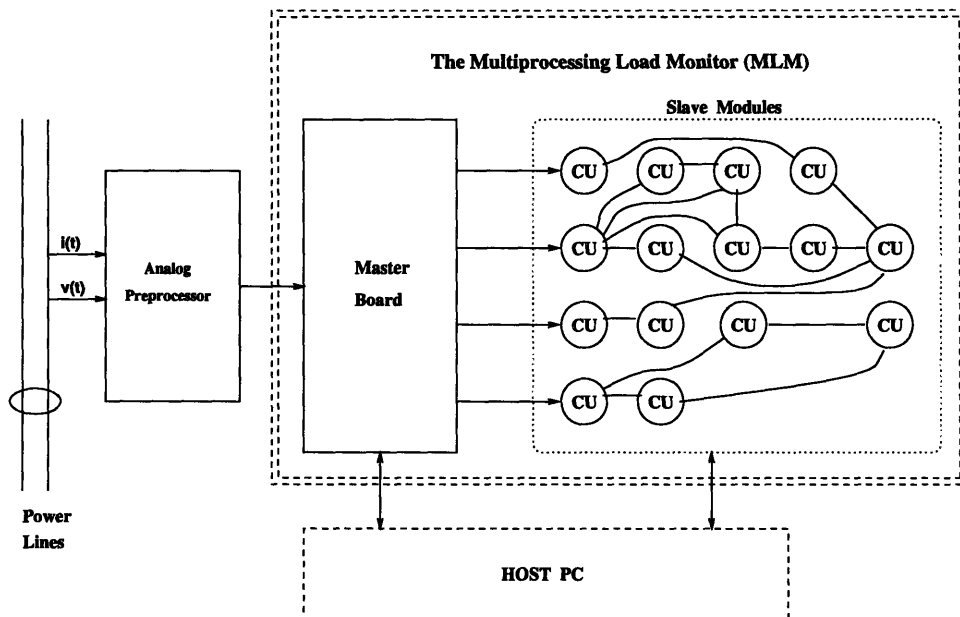


Figure 1.3: The Multiprocessing Load Monitor (MLM)

on. Thus, for instance, tradeoffs between computational complexity and reliability of event detection were considered. In the context of this study, an abstract model of a multiprocessing architecture for transient event detection was formulated. The **Multiprocessing Load Monitor (MLM)**, as it is referred to throughout this work, was designed to fulfill the three objectives discussed below.

A primary purpose of the MLM developed as part of this thesis, was utilitarian: to develop an inexpensive and commercially viable implementation of the multiscale transient event detector presented in [1]. To achieve this goal, an effort was made to parallelize the event detection and tree-structured decomposition algorithms proposed in [1], as discussed earlier. This parallelism was then reflected in the machine model (see Figure 1.3) which consisted of **front-end preprocessing blocks** (the analog preprocessor and the master board — see Chapter 3) and a mesh of **computational units (CU)**, programmable and configurable to allow maximum flexibility of function. The event detection and tree-structured decomposition operations could be distributed over these units in a way that was deemed analytically and/or experimentally optimal. Cost-effectiveness in the physical implementation was achieved by using inexpensive microcontrollers and associated memory to form the computational units.

Transient event detection is implemented on the MLM by distributing the functions of event detection, tree-structured decomposition, and result collation over several computational units. In addition the v-sections associated with the loads must be assigned to the computational units, keeping in mind processing power and interconnection limitations. An important direction for future work would be to analytically determine the optimal distribution of these various functions and load transient assignments over the computational units. Hence, one aim of the MLM is to provide an experimental platform that may one day be used to test the theory of optimal implementation of the multiscale event detector algorithm in a parallel processing environment. This demanded that the architecture of the basic computational unit be flexible with regards to both intercommunication and individual functionality. A unit should, therefore, be configurable to allow communication with the master board, the host PC or another computational unit, as shown in Figure 1.3. It should also be able to function as a pattern recognizer, perform tree-structured decomposition, or gather and interpret results from the pattern recognizers.

Finally, the MLM was designed to form the basis for a platform for nonintrusive diagnostic evaluations of industrial loads. The MLM allows raw data to be shipped — in bulk — directly to the host PC, while reporting all events detected. Diagnostic routines on the PC could use this information to detect changes in the transient patterns (e.g. in the shapes and relative placements of v-sections) that may indicate potential problems with the loads being monitored. Problems could be detected and remedied before they lead to significant down-time.

1.3 Thesis Outline

In this chapter we introduced the concept of nonintrusive load monitoring and the versatility of the Transient Event Detector (TED) of [1]. A preliminary discussion of the rewards of taking a parallel approach to the sequential algorithm of the TED is made. This underscores the goal of the thesis: To design and implement a prototype of this parallelized TED algorithm in the form of the Multiprocessing Load Monitor (MLM).

Chapter 2 explores the parallelism in the TED algorithm. Tasks that may be subdivided, distributed over different processors, and performed in parallel are identified and a parallel version of the algorithm is constructed. A discussion on the tradeoff between

the time/memory complexity and reliability of event detection is presented. In the context of the parallelized algorithm and complexity issues, an abstract model of the MLM is presented.

Chapter 3 describes one of the two main subsystems of the MLM, the **Master Board**, which is the **data acquisition front-end**. The data acquisition module's functionality and design methodology are explained, followed by a detailed report of the design and implementation of the module.

Chapter 4 rounds off our discussion of the hardware implementation of MLM by presenting the **Computational Units** of the MLM. Their function and top-level design are described, and supplemented with details of the actual implementation.

In Chapter 5, the software architecture of the MLM is elaborated. The code written for the computational units is described in the context of the parallel TED algorithm. The other major software effort was the design of the Host PC User Interface. Its functions and software implementation are also discussed in detail.

Chapter 6 presents the final configuration of the prototype MLM developed for this thesis. Results obtained during testing, and the performance achieved, are stated and evaluated with reference to the goals of this research.

Finally, Chapter 7 reiterates the aim of the thesis and gives a summary of the results achieved. It also indicates directions for future work based on the present research.

Several appendices are included at the end to support and enhance the body of this thesis. Appendices A and B consist of the schematics for the MLM and are important references for Chapters 3 and 4 respectively. Appendix C contains the PAL source code for both the data acquisition board and the computational modules. Appendix D lists the source code for the computational units. Appendix E gives details of the Host PC Interface software. Both ECM code and RISM code (resident in slave Read Only Memory) are included. Appendix F describes the design and construction of the PC I/O card used for MLM control. In Appendix G we give full details of the analog preprocessor by including [8] in its entirety. Appendix H supplements the discussion of prototype performance in Chapter 6. It gives details of the v-section sets for the monitored loads, as well as the software routines that activate the test loads in sequence. And finally, Appendix I is intended to be a pictorial guide to the construction of the prototype MLM.

Chapter 2

The Structure of the MLM

In this chapter we formulate the structure of a Multiprocessing System for transient event detection. We begin by analyzing the TED algorithm of [1] presented in Chapter 1, for data dependencies between its operations. Ensuring that these ordering constraints are met, a parallel version of the algorithm is developed. Before mapping this algorithm into a computation model, we discuss complexity and reliability tradeoffs. Finally, the structure of the Multiprocessing Load Monitor (MLM) designed and developed for this thesis is presented in Section 2.3.

2.1 Parallelism in the TED Algorithm

As discussed in Chapter 1, the TED algorithm of [1] monitors load activity by searching for transients in several data signals (e.g., real and reactive power) over several time scales. Consider the general case of searching for transients in data streams $1..R$, over time scales $1..S$. Transient event detection would consist of the following sequence of operations:

1. Sample data on each input stream [R operations].
2. Scale each stream in time, to get $S - 1$ new streams for each original input stream [$R(S - 1)$ operations].
3. For each data stream (on each time scale), perform transient event detection [RS operations].
4. Collate results and perform intra-scale v-section lockout on results for each time scale [S operations].

5. Perform inter-scale v-section lockout on results from all time scale [1 operation].

The total number of sequentially performed operations, O , is given by:

$$O = R + R(S - 1) + RS + S + 1$$

$$\Rightarrow O = 2RS + S + 1$$

Thus, for instance, if we were monitoring 5 input signals and searching over 3 time scales, a total of 34 operations would be sequentially performed. Most of these operations are not constrained to precede or follow one another. As a first step in bringing out the parallelism in the transient event detection algorithm, we must determine the *data dependencies* for the functions performed in the algorithm.

2.1.1 Data Dependencies for TED Operations

We begin by explaining the terminology that will be used in the discussion to follow. Suppose an operation g cannot occur until another operation f has completed. We denote the start of g by g^s and the end of f by f^e . The *precedence relation* [6] between f and g can be expressed as:

$$f^e \preceq g^s,$$

which is an inequality between the instants of the end of f and the start of g . This precedence constraint can be extended to cover entire operations by stating:

$$f \preceq g,$$

which expresses the fact that f and g cannot proceed concurrently. The precedence constraint \preceq is *transitive*, meaning that:

$$f \preceq g \ \& \ g \preceq h \Rightarrow f \preceq h$$

A *precedence graph* is a directed graph that represents the ordering on a set of events induced by precedence constraints. An arrow (directed edge) in the graph between vertices representing operations f and g specifies the constraint $f \preceq g$.

One of the main sources of precedence constraints is *data dependency*. A data-dependency constraint between operations f and g occurs if some value produced by f is required as input to g . A *data-dependency graph* is a precedence graph depicting only constraints due to data dependencies. It is an important visual tool that we shall now use to impose an ordering on the TED algorithm.

Figure 2.1 shows the data-dependency graph for the TED operations, for a sample case examining two time scales. The following data dependencies are present in the TED algorithm:

- For each stream independently: Data Acq. \preceq Event Detect. (original scale)
- For each stream independently: Data Acq. \preceq TS Decomp.
- For each new scale independently: TS Decomp. \preceq Event Detect.
- For each time scale independently: Event Detect. \preceq Intra-scale lockout
- Over all time scales and all streams: Intra-scale lockouts \preceq Inter-scale lockout

Perhaps more importantly, the data-dependency graph states that the following operations are *independent*.

- The operations of data acquisition, tree-structured decomposition, and event detection can be performed on each data stream independently.
- For a given time scale:
 - All result collation and intra-scale lockouts may be performed independently.
- For a given data stream:
 - All tree-structured decomposition operations can be performed independently.
 - All operations (event detection, collation and intra-scale lockout) on different time scales may be performed independently.

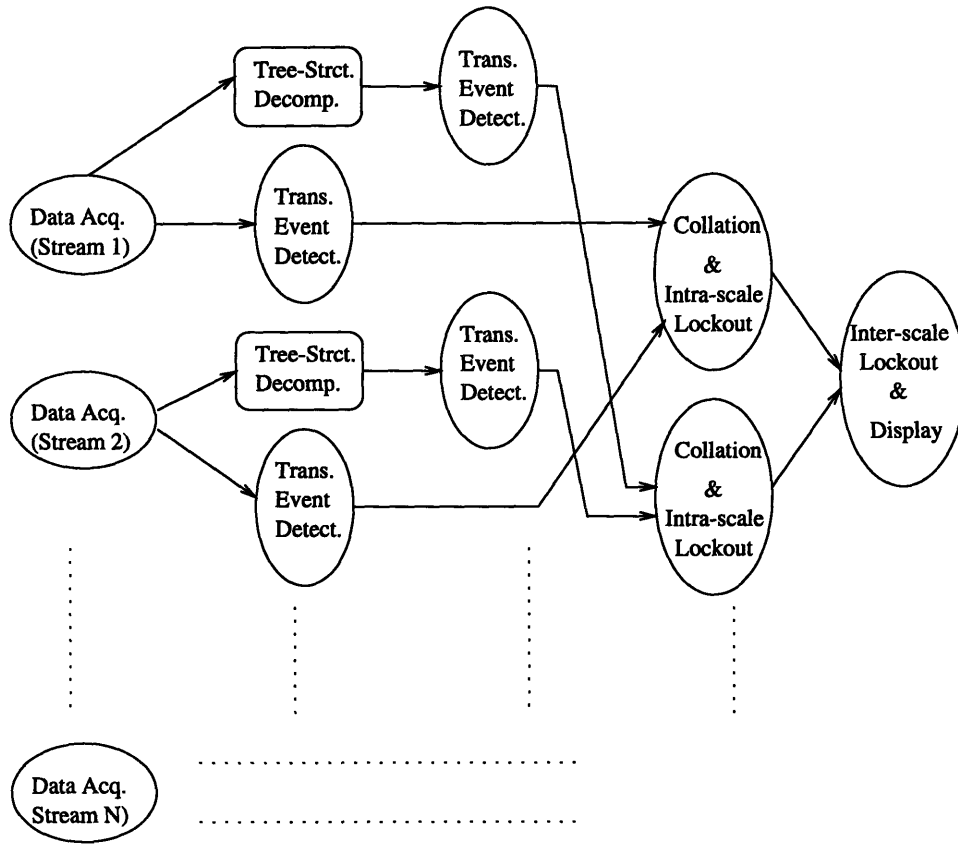


Figure 2.1: Data-Dependency Graph for TED Operations

2.1.2 The Parallel Transient Event Detection Algorithm

The data dependencies and “independencies” brought out in the previous section give us the guidelines by which to parallelize the transient event detection process. The Parallel Transient Event Detection Algorithm is given in Figure 2.2.

Here are the main steps in the algorithm:

1. Data acquisition is performed on all input streams in parallel.
2. The following operations are performed in parallel:
 - Event detection on the original time scale is performed on all data streams in parallel. Each load’s transients on every stream are searched for in parallel. In addition, the v-sections for a given load on a given stream may be searched for in parallel.
 - Tree-structured decomposition is performed on all the streams in parallel.

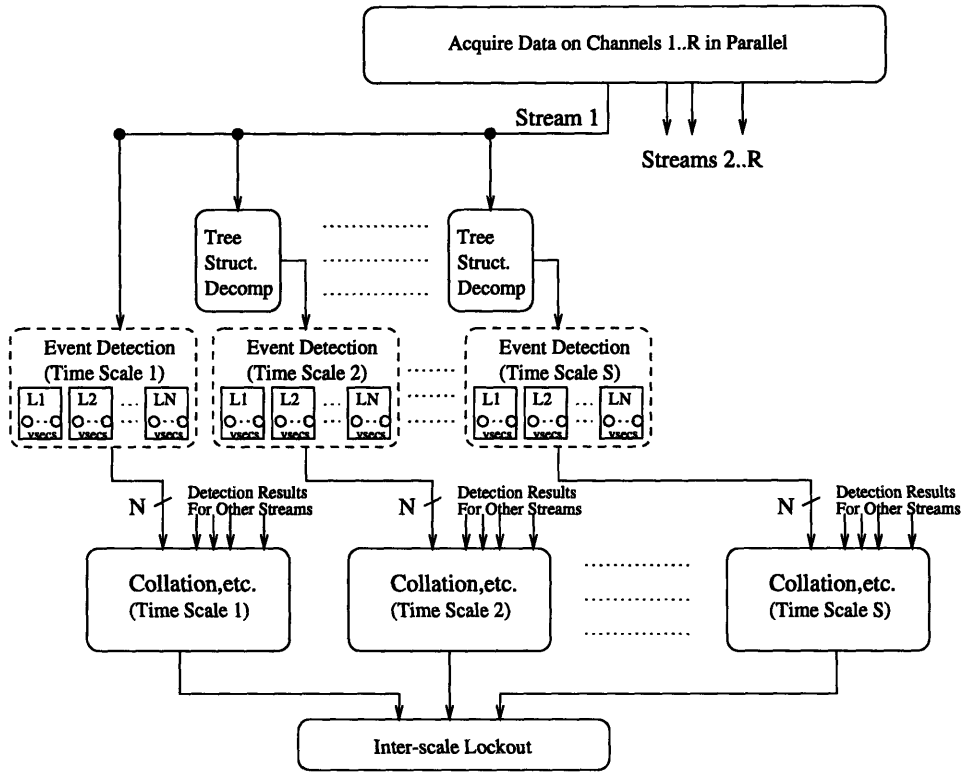


Figure 2.2: The Parallel TED Algorithm

3. Event detection on all the time-scaled data streams is performed in parallel in the same manner as for the original data streams.
4. Result collation (including intra-scale lockout) is performed on each time scale in parallel.
5. Inter-scale lockout is performed on the results of the collaters.

2.2 Complexity/Reliability Tradeoffs

There is considerable freedom in mapping the parallel algorithm into an abstract model for a multiprocessing transient event detector. In producing a practical, reliable, and economically viable implementation, there are restrictions in addition to the precedence constraints of the algorithm. Thus, before we set up the MLM model of choice, we cover the issue of complexity and, in particular, the tradeoffs between complexity and reliability of operation.

As our discussion is geared towards designing an MLM model, let us begin by stating

the design features that arise naturally from the parallel TED algorithm. The task of digitizing data over several channels and storing and relaying data suggests the use of specialized hardware. Instead of using one or more processors for data acquisition, a separate hardware module with specialized analog-to-digital converters, etc., should be designed to serve as the acquisition front-end. The sampling rate would be determined by the frequency characteristics of the inputs. The digitized data would be buffered and periodically shipped to the processors. The size of the transfer blocks and the sampling rate would determine the processing time available to the computational units between consecutive data arrivals. Thus, block size is related to the processing power of the units. The tasks of v-section search, tree-structured decomposition and result collation (with intra-scale lockout) would be distributed over a mesh of computational units, as stated earlier. Inter-scale lockout, the last step in the TED algorithm, may be performed by the host PC that would be needed to coordinate the MLM and provide a user interface. This last task assignment follows from the fact that inter-scale lockout requires significant memory (it requires access to information on *all v-sections*, their relative locations and their relative priority) and it need not be performed in real-time.

The mapping of loads and their v-sections over the processors and time scales available remains to be established. First we must decide on a processor that will give us adequate (yet inexpensive) computational power to perform the TED operations within the time frame established by the input sampling rate and the transfer block size. Another decision to be made is to determine the degree of parallelism to be incorporated in the final design: Just because two operations *can be done* in parallel does not mean they *have to be done* in parallel. This essentially means determining how many v-sections and how many loads should be sequentially processed by a single processor, as well as how many input streams a processor should search over. Finally, we must also decide which time scale a v-section should be identified on. A rigorous analytical solution to these questions is beyond the scope of our treatment of these issues. However, the factors that must be considered in determining the optimal MLM configuration are discussed here, and the architectural decisions made for the MLM (e.g., choice of processor and distribution of v-sections) are justified in the context of this discussion.

The most important factor governing the implementation of the parallel TED algorithm as a multiprocessing machine, is cost (remember that the goal of our work is to design

an *economical* and commercially viable load monitor). An increase in complexity is most unattractive to the extent that it raises cost. Our discussion of complexity is made under this simplifying assumption.

System Complexity for the MLM may be measured in terms of software complexity (or computational complexity) and hardware complexity (or *component count*). Software complexity itself represents the *computation latency* for a software operation as well as its *memory requirements*. *Component count* is an important factor as it directly affects cost. *Computation latency* of operations influences our design by helping us estimate the computation power needed of our processor. Moreover, the latencies of the various operations in the TED will determine how many tasks may be performed by a single processor within the established time frame. Higher latencies would mean adding more processors to perform all the operations, raising component count and hence, cost. *Memory requirements* are in a way the least important of the three factors: As long as the memory available in each processing module suffices, the memory requirements of an operation are irrelevant. The main influence on our MLM Model of the TED operations' memory utilization is in estimating the maximum memory that would ever be needed per computational unit. This would be used to define the memory range that must be provided by our processor. Once that is achieved, this part of system complexity may be ignored. Thus, as cost is our primary concern, for our purposes, system complexity is reflected and represented to a good approximation by component count alone.

In selecting a processor for the MLM we used the software complexity of the TED operations as our guide. First of all, memory requirements were considered. A processor searching for several v-sections over an input data block requires, in general, far less than 10K of data memory and less than 10K of program memory. A 16-bit addressing architecture, giving access to 64K of memory was therefore decided upon. Since flexibility in inter-processor communication would be required, access to several I/O Ports and individual I/O pins, a serial port, versatile interrupt handling, and other features characteristic of an embedded controller were needed. The Intel 80C196KC appeared to fit the bill.

Computation latencies of the various TED operations, e.g., pattern search, were used to verify this choice. This required determining the block size of input data transfers. As a first step the range of possible sampling rates was fixed. For the types of inputs that the MLM would work on, the sampling rate would be between 100Hz and 250Hz. Given

the interrupt processing latencies of the 80C196KC, a block size of one (i.e., sending each sample as soon as it is acquired) would be unrealistic — too much time would be lost processing interrupts a few hundred times each second. If a very large transfer block size is chosen, say 5000 samples, new data would be shipped after 25 seconds (given a sampling rate of about 200Hz). This is impractical as it would make load identification lag the load activity by nearly a minute, if a v-section is split between two transfer blocks. Such a large block size would also be unsuitable because of the long transfer time. The correct size is somewhere between these extremes. The most important clue is the computation time of the TED operations on the 80C196. Experiments were conducted with various v-section sizes (varying from 10 to 200 points) on data blocks varying in length from 100 to 2000 sample points. Block sizes in the range 100–1000 points emerged as good choices. These gave the processors 0.5–5.0 seconds between consecutive data arrivals, and processor interrupt frequency was not unreasonable. We chose a block size of 500 points. The time needed to carry out pattern search using Euclidean filtering (see Chapter 4) on this block size for various template sizes was measured. The results showed that a 100 point template could be searched in about 0.75 seconds. Hence, the biggest template that could be searched would be about 325 points long. It was also seen that the total points searched determined the computation time, regardless of the fact that the points comprised a single v-section template or spanned several templates. Thus, five 30 point templates took about the same time (slightly more due to initialization overheads, etc.) as a single 150 point v-section. As it was estimated that actual v-section templates would vary in size from about 10 points to 35 points on average, this meant that about 10–25 v-sections could be identified on a single 80C196 in the allotted 2.5 seconds. Tree-structured decomposition required on the order of 0.3–0.4 seconds, which meant that a processor performing decomposition could double as an event detector. This timing data guaranteed us the flexibility in load and v-section assignment that we needed. The 80C196 was therefore judged a feasible choice for the given sampling rate and transfer block size.

Having established the time frame for processor computation, we now comment on the degree of parallelism needed in our model. Since we transfer data every 2.5 seconds, the worst-case lag between the *end* of a load transient and load identification being reported¹ is

¹Since the durations of transients vary from load to load, we measure identification lag from the *end of the transient* to the load being identified by the MLM.

just under 5 seconds. It therefore makes little sense to distribute one v-section per processor i.e., *completely* parallelize pattern search for load v-sections. Not only does this waste each processor's computation power, it makes no difference to the worst case delay in reporting load activity, while tremendously increasing component count. A much better scheme would be to allow some sequential processing per processor e.g., by allotting 5 to 10 v-sections to each event detector processor.

Component count could reasonably be increased if it leads to increased reliability in the functioning of the MLM². The MLM would be deemed *perfectly reliable* if a load turn-on was reported if **and only** if the load turned on. That is, the MLM must never miss a load turning on and it must not be deceived into falsely reporting load activity when none occurred. The problem with transient event detection (and with pattern recognition in general) is that the events — i.e., the v-sections — are not perfectly repeatable, are accompanied by background noise, and may occasionally be partially distorted due to high load activity in the monitored environment. The error threshold set must therefore be large enough to accommodate all possible v-section manifestations. This means transients due to other loads turning on or off may be seen by the discriminating filter to lie within the error bounds, so that a false identification may occur. We consider below some techniques for overcoming these problems and note how they inevitably require a complexity tradeoff.

One way to increase reliability would be to use a more sophisticated discriminating filter, or filters, to perform the pattern search. The hope would be that by the maximal use of the information available, these techniques may be able to define the tightest error threshold for positive recognition. The problem with this scenario is the necessary increase in software complexity (and, in particular, computation latency). This in turn would mean an increase in component count. In our case, the Euclidean Filter (see Section 4.2.1) was chosen for pattern discrimination largely because it was compatible with the processing power of the 80C196 — a more sophisticated filter might severely reduce the number of v-sections that may be processed per computational unit, leading to an unaffordable increase in complexity. However, more research is needed to determine the ideal filter in terms of resolution per computational complexity, for a given set of resources (such as processing power, allowed

²The term reliability is used here to mean correctness of operation — i.e., how reliable is the MLM as a load monitor. It must not be confused with system reliability (in the context of robustness and fault tolerance) which is defined in terms of *the mean time to failure of the system as a whole* [7].

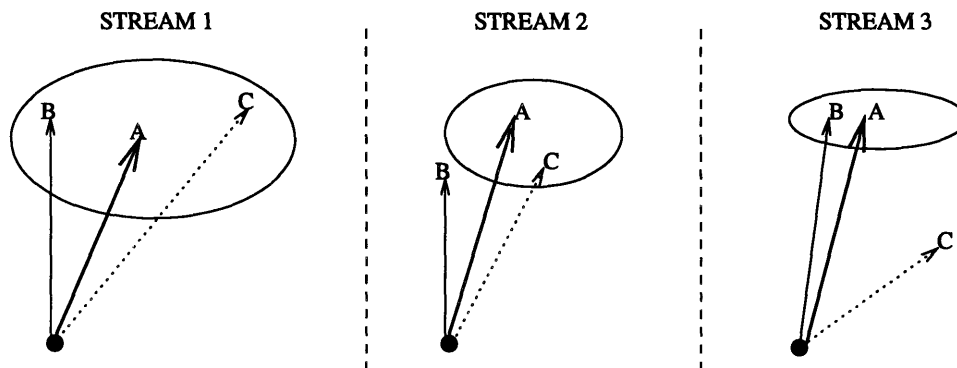


Figure 2.3: Improving Reliability By V-section Search Over More Streams

time, etc.).

We could also improve MLM reliability by searching for more v-sections over more streams, in the identification of a load. Figure 2.3 shows the case of identifying load A, with the challenge of loads B and C giving turn-on transients that resemble the transient for A in certain input streams. The ovals represent “identification spaces” set up by the error thresholds for the transient sections of load A on each stream. The dark arrow represents a transient in the input stream due to the turning on of load A, the dotted arrow represents transients in the input due to load B, and the light arrow denotes transients due to load C. Note that the error threshold for load A in stream 1 is large enough to allow transients due to load B or load C to be occasionally interpreted as load A transients. Thus, if pattern search for load A is conducted only on stream 1, false identifications of A will occur. Load A has a transient v-section in stream 2 and if this is used in conjunction with stream 1’s transient, we see that transients in stream 2 due to load B never make it to A’s “identification space”. In this way, false identification of load A due to B’s activity is eliminated. Similarly, v-section search in stream 3, eliminates spurious reporting of load A turning on, due to load C’s activity. Clearly, reliability is improved and, equally clearly, unfortunately, so is component count. Nevertheless the MLM model should allow the use of this technique to improve reliability.

An important issue in the context of reliability and complexity trade-offs is determining the best time scale on which to search for a v-section. If a v-section search is conducted on the finest scale possible (i.e., the highest sampling rate), we have more frequency contents of the input signal available (higher resolution) than at any other scale: To go to a coarser

scale requires downsampling (including pre-filtering), in which the higher frequencies of the input stream would be lost [19]. Hence, reliability is maximized at the finest scale. However, this also means more points per v-section, which increases memory requirements and computation time, and consequently component count. Downsampling the input data and searching for a proportionally smaller v-section would save on processing complexity but may reduce the resolution of transient recognition. To strike the perfect balance would require studying the frequency characteristics of the v-section, the possibilities of a false hit, and the computation power available. In general, if the variations in the transient are slow, or scaled out in time — i.e., the input is predominantly a low frequency signal — so that no useful information is lost in the process of low-pass filtering and decimation, this procedure should be undertaken, and the v-section identified on the coarser scale. If the v-section varies quickly in time (high frequencies present), downsampling could affect identification reliability. Thus, tree-structured decomposition and identification on a coarser scale is a particularly good idea for v-sections that exhibit slow variation over an extended period of time.

It is possible to improve reliability without particularly increasing complexity. An example is the idea of having individual error thresholds for each v-section searched. Certain loads exhibit remarkably consistent transient behaviour on certain input streams. Other loads may not be as cooperative. If there was one global error threshold, reliability of load identification would be severely decreased. Hence, we should customize the error threshold for each pattern to be searched according to the degree of its repeatability. The increase in software complexity is minor as far as computation time goes, and not too costly in the extra memory needed.

The discussion in this section focussed on issues governing the optimal configuration of a multiprocessing machine for load monitoring via transient event detection. It was also intended to ground our choice of implementation of the parallel TED algorithm in reality. As stated earlier, much theoretical work needs to be done in this direction. The good news is that the MLM's design allows easy reconfiguration. This would therefore provide the ideal platform to test the optimal distribution of these various functions and load transient assignment over the MLM processors.

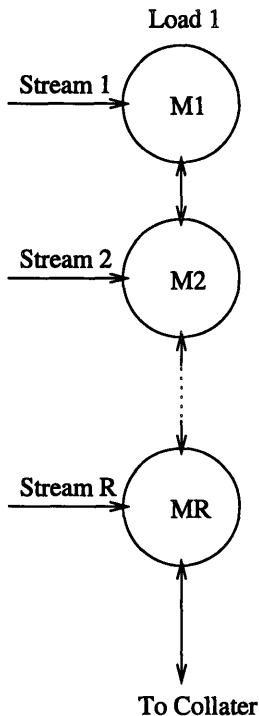


Figure 2.4: Identifying a Load on One Time Scale

2.3 The Structure of the MLM

The MLM model is now presented in the context of the TED dependency graph and the complexity issues raised in Section 2.2.

Consider once again the parallel TED algorithm in Figure 2.2. A natural way to structure the MLM would be to have a separate subsystem for data acquisition, feeding a network of processing modules performing the rest of the operations. The data acquisition front-end would sample all input channels simultaneously. It would buffer the digitized data and periodically ship data blocks to the processing modules. This way the sampling and transference of input data are performed in parallel over all input channels.

The allocation of tasks over the processing modules follows naturally from the algorithm and the complexity constraints:

- Each processing module works on only one input stream.
- Each processing module is assigned exactly one of three functions:
 1. Tree-structured decomposition.

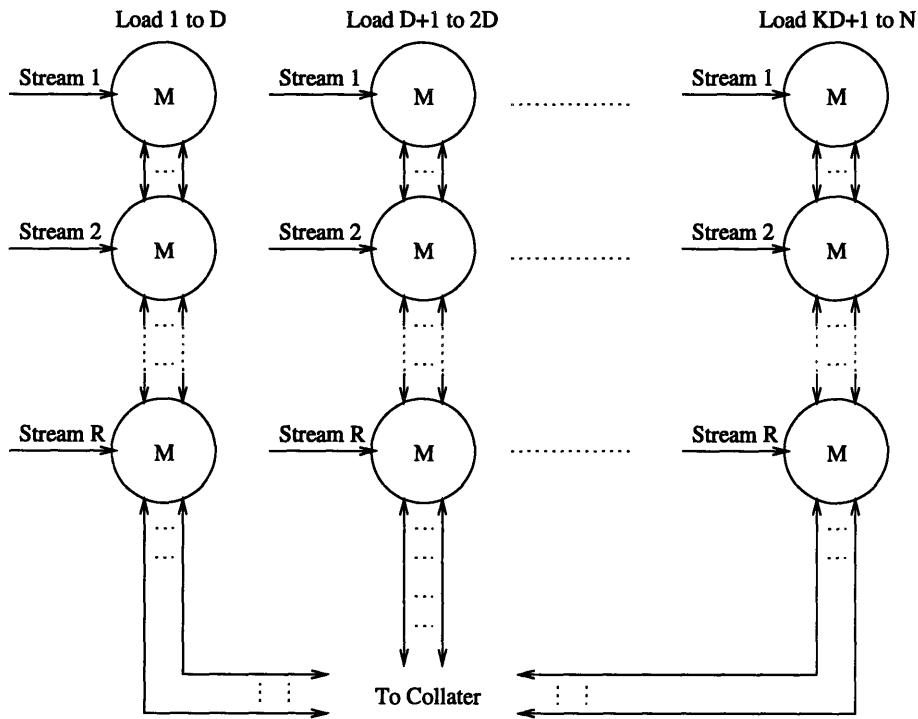


Figure 2.5: Identifying N Loads on One Time Scale

2. Searching the input stream for v -sections belonging to one or more loads.
 3. Collation of event detection results on a given time scale.
- Inter-scale lockout is performed by a host PC.

In keeping with our goal of a practical and economical implementation, we do not choose the “finest grain” distribution of tasks over the modules. Hence, while v -sections of a load on a stream may be distributed over different processors and identified in parallel, we choose to search for all v -sections on one data stream sequentially on one processor. Moreover, multiple loads may be assigned to a single processor. These decisions are not dictated by precedence constraints. Rather they reduce component count (fewer processors) and communication complexity, while not significantly affecting computation time.

Consider the task of identifying one load on one time scale. Suppose the load has transient v -sections on streams 1 through R . Figure 2.4 shows the configuration of modules needed. Module M_1 searches stream 1 for the v -sections present in that stream. Upon detecting all v -sections, it signals M_2 , which has been searching for v -sections in its own input stream. The detection results are thus passed down this linear chain. When all modules

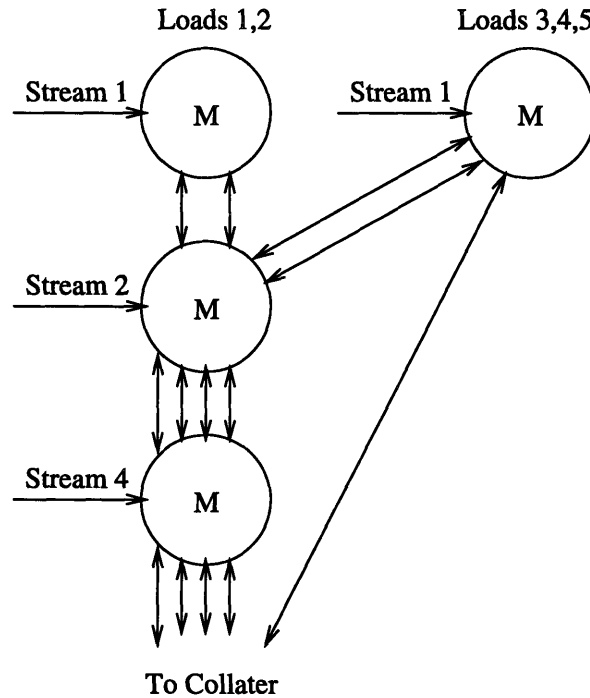


Figure 2.6: A More Compact Load Assignment

including the final processor MR, have identified their v-sections, the load is considered identified on the given time scale and a message to that effect is passed on to the collater for that time scale.

We now expand our example to cover the identification of N loads on a particular time scale, with v-section search being conducted on all R streams. Figure 2.5 shows the general processing module configuration. Each processor searches for v-sections of D loads on their respective input streams. Whenever a processor identifies all the v-sections for one of its loads, it relays a message to the next processor in the column. As multiple loads are being tracked, it is important for processors to communicate which load's v-sections have been identified. Note that the column-like placement or linear linking of modules, with communication possible only with neighbors, is an economical configuration which easily serves our purpose. Details of how the communication along these chains of processors is implemented are given in Chapter 4.

The symmetrical distribution of loads and their v-sections is somewhat misleading. In reality most loads will *not* have v-sections on all streams. They will also not have the same number of v-sections or similarly sized v-sections on different streams. The distribution and

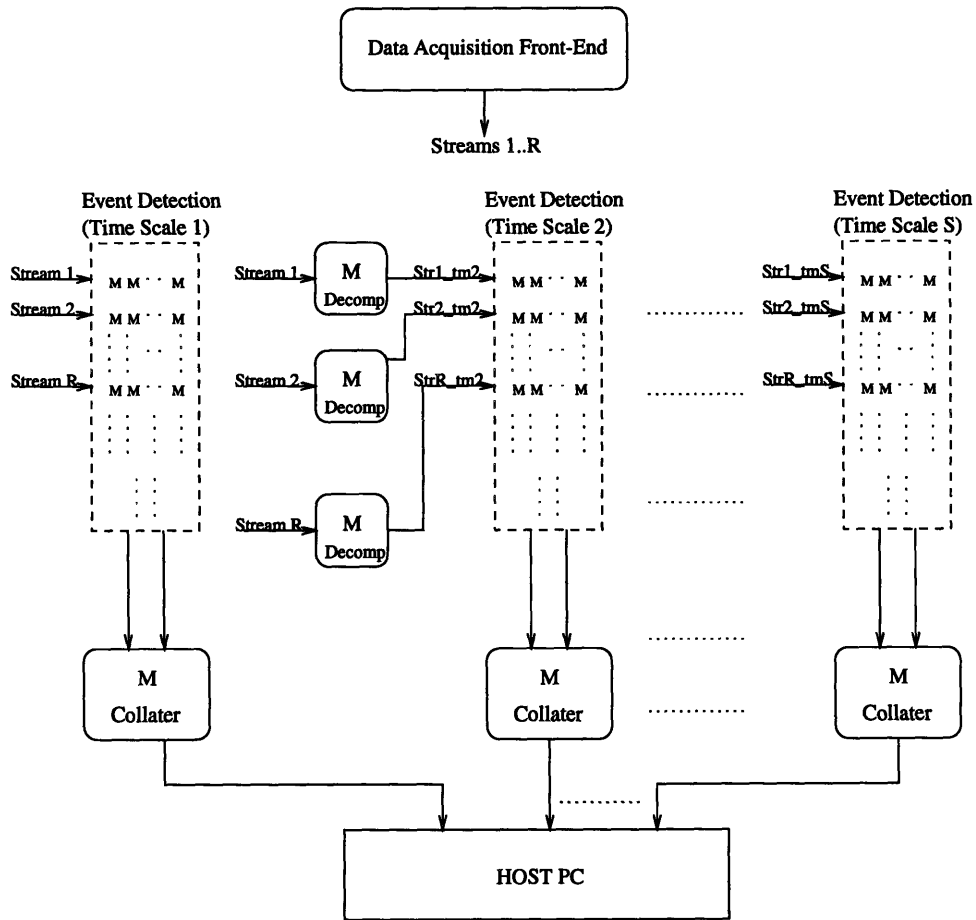


Figure 2.7: The MLM Model

nature of v-sections will also differ from load to load. Thus while the columnar structure of 2.5 is a good visualization, a more compact configuration will be possible in most practical instances. Figure 2.6 shows the “custom” distribution of loads and v-sections across processing modules. Loads 1 and 2 have a large number of v-sections in stream 1 compared to loads 3,4, and 5. Hence, one processor is dedicated to v-sections of Loads 1 and 2 in stream 1. Another looks for stream 1 v-sections of Loads 3, 4, and 5. Loads 1 through 4 have fewer/smaller v-sections in streams 2 and 4. Load 5 has no v-sections in any stream other than stream 1. These facts are used to completely identify all five loads over four processors, as shown in Figure 2.6.

Figure 2.7 gives the model of the Multiprocessing Transient Event Detector that will be implemented in this thesis and utilized for load monitoring. A data acquisition front-end samples all input streams and relays the data blocks to the processing modules. Tree-

structured decomposition is performed on each stream independently. Event detection on the original time scale begins as soon as the front-end sends new data. Event detection on other scales is performed following the completion of tree-structured decomposition of the input data to that scale. Collation and intra-scale lockout are performed independently on each time scale. The task of inter-scale lockout is relegated to a host PC which also controls the entire MLM by downloading code to the processors and uploading results from them. The configuration of modules for event detection on each time scales is load-dependent. Details of this configuration are irrelevant in the abstract model.

Chapter 3 describes the hardware design and implementation of the data acquisition front end. Chapter 4 follows up with the design of the slave modules and details of how inter-slave communication takes place. Chapter 5 concludes the implementation details of the MLM by discussing the software design for the entire system including the software development of the host PC interface.

Chapter 3

The Data Acquisition Front-End

A major contribution of this thesis is the development of a multiprocessor-based nonintrusive load monitor. The **Multiprocessing Load Monitor (MLM)** consists of two distinct subsystems: the data acquisition front-end, or the **Master Board**, and the main computational unit, the **Slave Module**. In addition, an **Analog Preprocessor**¹ interfaces the MLM with the power load lines by providing conditioned analog inputs to the master board.

In this chapter we focus on the function and design of the data acquisition front-end. We begin with a brief discussion of the analog preprocessor and the inputs to the master board.

3.1 The Analog Preprocessor

The analog preprocessor is the primary interface between the digital world of the MLM and the *utility service entry*, i.e., the power lines driving the loads being monitored. Its principal outputs are estimates of the envelopes of real and reactive power, as well as in-phase and quadrature third harmonic contents of current. It also gives the higher harmonic contents of the current, providing 16 analog channels per phase, to be analyzed by the MLM computational units. Full details of the analog preprocessor are given in [8]. This paper is included in its entirety in Appendix G.

Given below is a listing of the outputs of the analog preprocessor. Note that in-phase and quadrature harmonics are referred to with the same P and Q notation as “real power”

¹The Analog Preprocessor was designed and developed by S.B. Leeb and S.R. Shaw. See Appendix G for details of its functionality.

and “reactive power”, even though there is generally no higher harmonic of voltage and so, strictly speaking, no high harmonics of power. The nomenclature used here is adopted from [8], and is simply a “short-hand” notation for describing the in-phase and quadrature higher current harmonics.

1. Envelope of real power, P .
2. Envelope of reactive power, Q .
3. Envelope of second harmonic of real power, $2P$.
4. Envelope of second harmonic of reactive power, $2Q$.
5. Envelope of third harmonic of real power, $3P$.
6. Envelope of third harmonic of reactive power, $3Q$.
7. Envelope of fourth harmonic of real power, $4P$.
8. Envelope of fourth harmonic of reactive power, $4Q$.
9. Envelope of fifth harmonic of real power, $5P$.
10. Envelope of fifth harmonic of reactive power, $5Q$.
11. Envelope of sixth harmonic of real power, $6P$.
12. Envelope of sixth harmonic of reactive power, $6Q$.
13. Envelope of seventh harmonic of real power, $7P$.
14. Envelope of seventh harmonic of reactive power, $7Q$.
15. Envelope of eighth harmonic of real power, $8P$.
16. Envelope of eighth harmonic of reactive power, $8Q$.

The master board accepts 8 analog inputs from the 16 choices. In a typical case, the master board would look at P , Q , $3P$, $3Q$, $5P$, $6P$, $7P$, $8P$. These inputs would be sampled and the data passed to the slave modules, which would search this data for transient events.

3.2 Functional Overview of the Master Board

The Master Board acts as a digitizer, a buffer and a relay between the analog preprocessor and the computational units, as well as between the preprocessor and the host PC. It is proclaimed *master* because it has the highest priority in the eyes of the computational units (its *slaves*) when it informs these units that new data is ready to be transferred to them. Moreover, because of the single-sided handshaking between the master board and the slave modules, the master does not wait for the slaves to announce their availability, imperiously beginning the data transfer after a fixed time period.

The master board is therefore, the data acquisition front-end of the MLM. It interfaces with the analog preprocessing module by accepting 8 analog channels from it, digitizing the signals, storing them in RAM, and periodically transferring the data blocks to the slave modules. It also interfaces with the host PC via a PC I/O card, and provides the PC with an expansive window of acquired data upon request. Its primary functions are:

- Digitize the analog data coming from the preprocessor.
- Collect and store the digital data.
- Periodically transfer suitable size blocks to the slave modules.
- Ensure data integrity by not allowing any loss of incoming data during transfer of sampled data to slave modules.
- Store, and upon request, transfer, large windows of data to the PC.

3.2.1 Data Acquisition and Storage

Figure 3.1 shows the basic blocks of the master board. The eight analog input channels get serviced by two 4-channel Analog-to-Digital Converters (ADCs). Each A-to-D converter relays its conversion results to dedicated memory in the form of 8Kx8 SRAMs. Each ADC, the associated memory ICs, and its memory address counters form an Acquisition Bank. Within each bank, the acquisition of data and its storage into memory is governed by the Sampling Rate Generator Module and the Control Logic. The basic data path of the master board consists of an 8-bit data bus, a 12-bit address bus, and various control and status signals. An important control signal, the *Sampling Rate Signal* is generated by the sample

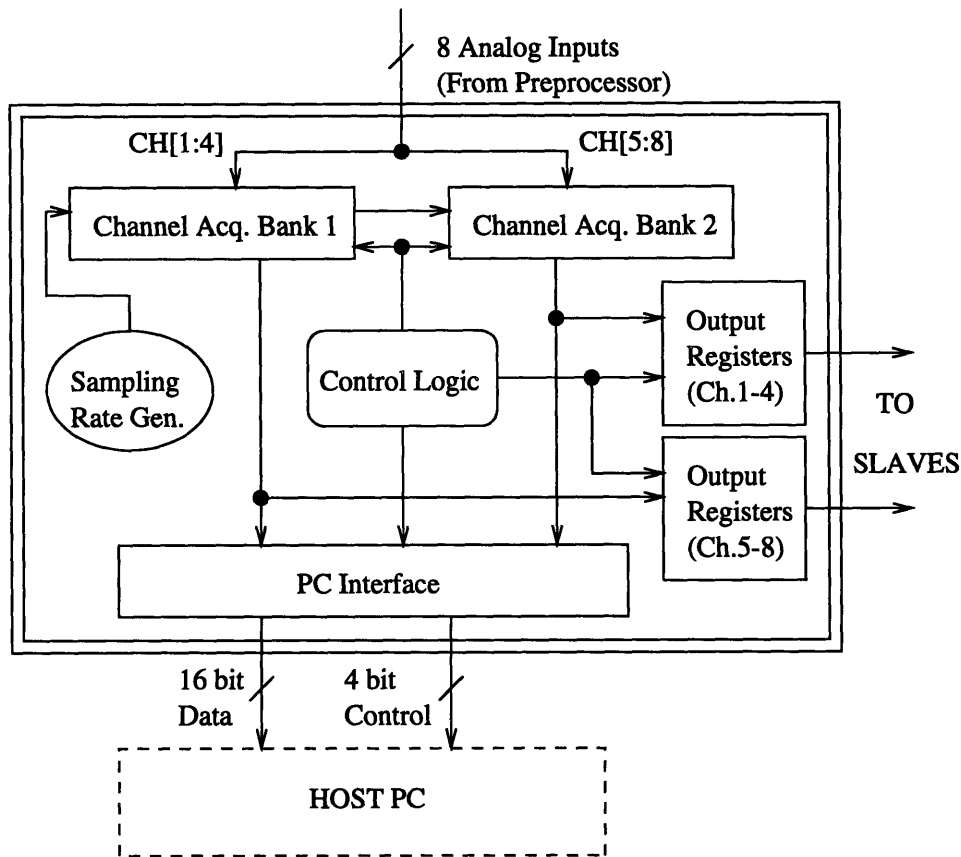


Figure 3.1: The MLM Master Board

rate generator. This is a square wave which goes to both ADCs and triggers them to sample the input channels. Its frequency, which may be selected via dip-switch settings, determines the sampling rate. After each analog-to-digital conversion, a signal is sent to the control logic informing it that new sampled data is ready. The control logic then reads the data from the ADCs and writes it into memory.

3.2.2 Data Transfer to Slave Processors

The acquired data must be periodically transferred to the slave processors. Transfer begins when the memory ICs are filled up. The master board issues an interrupt signal to all the slave processors. It allows the slaves to leave their computation activity and enter into a listening mode. Data is read sequentially from both acquisition banks simultaneously, and transmitted to the slaves. An “output” register is dedicated to the communication of data for each input channel. The output lines of the registers are connected to the input ports of

the slaves for that input channel. During data transfer, each of these eight output registers is loaded with a data sample from their channel. The control logic then generates *Data Available (DAV)* signals to tell the slaves that data may be read. This cycle is repeated until the entire data block is transferred.

In the present implementation, the block size is 512 samples (1024 bytes). The transfer of this large set of data cannot usually happen neatly between two sampling events in the ADC. If an ADC conversion is not read, new conversions will overwrite it, compromising the integrity of the sampled data. To avoid this, the Programmable Array Logic (PAL) ICs controlling the transfer of data and the PAL controlling the its acquisition, communicate with one another: If the ADCs signal the arrival of new data during a block transfer, the acquisition PAL “interrupts” the transfer PALs, gains control of the address and data buses, reads the new data into the right memory location, and returns the control of the address and data buses to the transfer PALs.

3.2.3 Host PC Communication

The main function of the master board is to transfer data to the slave modules. In addition, the data is shipped to the host PC to allow, among other things, comparison with the results of transient event detection produced by the slaves and to carry out high-level diagnostic checks on the operating loads. For this purpose, the master board incorporates a PC interface section with its own memory, output registers, connectors, and control PALs. The schematics for this section are given in Figure A.4, in Appendix A.

The memory in the PC interface is larger than in the acquisition banks — eight times larger, allowing a total of eight data blocks to be stored and upon request, transferred to the PC. A transceiver IC, the LS245, in the interface section acts as the *gateway* to the PC interface memory: If the transceiver is enabled, the data bus of the acquisition bank is connected to the interface section data bus. In this case, all writes to the acquisition bank memory also go through to the PC interface memory. This is the *storage mode*. During the *PC transfer mode*, the PC interface module is disconnected from the rest of the board and writes in the acquisition bank are not seen by the interface memory. Instead, the data collected in the interface memory is read out and transmitted to the PC. Once the transfer is complete, the interface module returns to the storage mode.

3.2.4 Control Logic

There is no on-board processor for the master board. Instead, the microcontrol of the various functional blocks in the master board is implemented by Finite State Machines (see [6]) programmed into six Programmable Array Logic (PALs) ICs. Details of these FSMs are given in Section 3.3. The PAL Code is listed in Appendix C. A summary of the master board PALs, and their functions, follows:

- **PAL_AD**: Controls the data acquisition process, including reading data from the ADCs into memory. Communicates with the *transfer PALs* (see below) to ensure no conversions in the ADCs are overwritten while a transfer is in progress.
- **PAL_TR1** and **PAL_TR2**: Control the transfer of data to the slave modules, providing the necessary handshaking signals. Communicate with **PAL_AD** to ensure integrity of acquired data.
- **CLK_GEN**: Manages the operations of the sampling rate generator unit.
- **PAL_PC1** and **PAL_PC2**: Govern the storage of data and its transfer (including performing the handshaking protocol) to the host PC.

3.3 Design and Implementation

Several functional subsections comprise the master board. Their data paths and logic-level details are given below.

3.3.1 A/D Conversion

The master board uses two **AD7874s** [9] for A-to-D conversion (see Figures A.1 and A.5). These Analog Devices IC's are complete 12-bit, 4-channel data acquisition systems with four track/hold amplifiers allowing simultaneous sampling of all channels. The ADC expects a negatively-asserted pulse on its \overline{CONVST} line. On the rising edge of \overline{CONVST} , all four input track/holds go from track to hold. Conversion is then performed sequentially on channels 1 through 4, and the results are stored in on-chip registers. Upon completion of all four conversions, the \overline{INT} signal goes low indicating data availability. Data is presented on the output lines **DB[0–11]**, when \overline{CS} and \overline{RD} are asserted. To read the conversion results

of all four channels, the \overline{CS} and \overline{RD} lines must be asserted four times and each time the data must be read from the port. Thus, reading data from the AD7874 consists of four read operations. The first read after a conversion always accesses channel 1's conversion result, the second read always accesses the second channel's data register, and so on. Note that \overline{INT} is deasserted (goes high) after the first read operation.

In the master board, two AD7874's are used to sample eight analog inputs. In order to sample all eight channels simultaneously across both ADCs, the \overline{CONVST} inputs of both ADCs are tied together and driven externally from the same source. Also, in order to ensure good full-scale tracking across the ADCs, the *REF OUT* signal of the first ADC is fed to the *REF IN* input of the second ADC. The \overline{CS} and \overline{RD} inputs are also tied together. The voltage V_{ss} ($-5v$), is provided by a common voltage regulator (79L05), and *AGND* and *DGND* signals are tied together at a single point close to each AD7874 to reduce noise. Both ADCs have *CLK* tied to V_{ss} so that the internal laser-trimmed clock oscillator is used for all on-chip operations. With this internal clock source, the maximum conversion time, from the rising edge of \overline{CONVST} to the final conversion (channel 4's result registered), is 35us.

The data outputs of each ADC go to two buffers (74LS245s), the six LSBs to one and the six MSBs to the other. These buffers have their outputs tied together and connected to the 8-bit data bus (the 2 most significant bits of which are not used). The data bus goes to the I/O lines of an 8Kx8 SRAM. During a read operation, when the AD7874 outputs data, each latch is enabled in turn (the Low Data "Byte" first) and written into the SRAM. Note that each AD7874 has its own pair of buffers and its own storage RAM. The conversion results of both ADCs are transferred to memory in parallel.

In order to control the data acquisition process, a 22v10 PAL is used. This control PAL, labelled PAL_AD in the schematics (see Figure A.1), and its operation are described in detail in Section 3.3.4. PAL listings are given in Appendix C.

3.3.2 Data Storage

The master board samples all channels, stores the data in RAM, and then transfers a block of data to the slave modules. For our purposes, a block size of 512 points was deemed optimal, each point occupying two bytes (12-bit A/D conversions). A total storage of 4K is needed per AD7874, as there are four channels per ADC. Hence, we used 8Kx8 RAMs [10].

These are addressed by HC4040's which are 12-bit synchronous counters with positively-asserted *RESET* inputs (see Figures A.1 and A.5). Both RAMs share the same counters. Two counters are used to address a RAM, one is labelled the "AD_COUNTER", the other is called the "TR_COUNTER". Each counter outputs its count to a pair of 8-bit buffers, the LS245's. The eight LSB's of the count go to one buffer, and the four MSB's of the count go to the other. The buffers drive the address lines of both RAMs. At a time, only one pair of buffers is enabled. Hence, only one counter addresses the two RAMs at any given time.

During the *acquisition* phase, the AD_COUNTER addresses the RAMs, i.e., its buffers are enabled and the buffers in front of TR_COUNTER are disabled. Each time a conversion is completed, eight bytes are written into each RAM (four sample points), and the AD_COUNTER is properly incremented by the control logic to provide correct memory addressing. When 512 conversions have been completed and 4K of RAM has been filled, the AD_COUNTER increments from 0x1111 to 0x0000. This signals the control logic that a window of data is ready for dispatch to the slaves. As a result, the *transfer* phase is entered. The AD_COUNTER buffers are now disabled and the TR_COUNTER buffers are enabled, so that memory is now addressed by TR_COUNTER. The sample points are read out of memory and sent to the slave modules via the Slave Interface Circuitry, detailed in the next section. If during the transfer phase, the ADCs assert \overline{INT} to signal a conversion, the control logic disables the TR_COUNTER buffers, re-enables the AD_COUNTER buffers and stores the samples at the addresses dictated by AD_COUNTER. It then passes addressing control back to TR_COUNTER. At the end of the block transfer, the AD_COUNTER buffers are enabled in place of the TR_COUNTER buffers, and the board re-enters the *acquisition* mode.

One important aspect of addressing the RAM must be noted. The RAM is read sequentially during the transfer phase but it is *not written sequentially* during the acquisition phase. The order in which the samples are read from the ADC is as follows:

1. LSB of Channel 1 Data.
2. MSB of Channel 1 Data.
3. LSB of Channel 2 Data.
4. MSB of Channel 2 Data.
5. LSB of Channel 3 Data.

6. MSB of Channel 3 Data.
7. LSB of Channel 4 Data.
8. MSB of Channel 4 Data.

This, however, is not the order in which we want the data to appear in RAM. As we will be retrieving data from RAM sequentially, it will be more convenient to place the four LSBs together and store the four MSB's next:

1. LSB of Channel 1 Data.
2. LSB of Channel 2 Data.
3. LSB of Channel 3 Data.
4. LSB of Channel 4 Data.
5. MSB of Channel 1 Data.
6. MSB of Channel 2 Data.
7. MSB of Channel 3 Data.
8. MSB of Channel 4 Data.

This ordering allows us to interleave dispatches of the data from different channels when we sequentially retrieve sample points. For instance, if slave module X processes Channel 1 data, the master board first sends it the LSB of the Sample point and lets it take some time to store away the data in its local RAM. In the mean time the master board dispatches the LSBs of Channels 2, 3, and 4 to other slave modules. By the time it retrieves the MSB of the Channel 1 data, module X is ready to receive this data, and no delay is needed. This interleaving saves time and simplifies data transfer addressing. To achieve this ordering, we must connect the AD_COUNTER output to the RAM address lines in the following order:

- AD_COUNT 0 → RAM ADDR 2
- AD_COUNT 1 → RAM ADDR 0
- AD_COUNT 2 → RAM ADDR 1

Thus, the eight writes to memory do not access sequential (0→1→2→3→...7) locations. Instead, the following sequence of memory accesses is observed:

Location 0→ Location 4→Location 1→ Location5→
→ Location 2→ Location 6→Location 3→ Location 7

It is easily verified that this access pattern produces the desired ordering.

3.3.3 Slave Interface

Eight 10x2 connectors line one edge of the master board. The outputs of eight 74F574s go to these connectors (refer to Figures A.3 and A.5). Each connector also receives two handshaking signals from the control logic. One row of pins on each connector is grounded. This assembly forms the Slave Interface Circuitry on the master board. Each analog channel thus has an LS574 and a 10x2 connector associated with it. The four 574s for Channels 1–4 take their input from the I/O lines of one RAM, and the four 574s for Channels 5–8 take their input from the other RAM. Obviously data for any one on Channels 1–4 and any one of Channels 5–8 can be accessed simultaneously, since they reside in separate RAM. This is precisely what occurs during data transfer. The control logic asserts the Data Available, or *DAV*, signals which are not only transmitted to the slave modules via the connectors, but also go to the *CLK* input of the interface registers (574s) and latch the respective data bytes on their rising edge. This way when the slave modules see *DAV* asserted, data is already latched at the corresponding 574 outputs and available, via the connectors, at the input ports of the slaves.

The control logic issues four Data Available signals: *DAV1*, *DAV2*, *DAV3* and *DAV4*. *DAV1* clocks the registers receiving data from analog channels 1 and 5, *DAV2* clocks the registers receiving data from analog channels 2 and 6, and so on. Thus, the data is read sequentially from both RAMs simultaneously and transmitted by the assertion of the appropriate *DAVs*, which are asserted in order from *DAV1* through *DAV4*. After one round of *DAV* assertions (*DAV1* → *DAV2* → ...*DAV4*), the LSB of a single sample point from *all eight channels* has been transferred to the slave modules. After two rounds of *DAV* assertions (*DAV1* → *DAV2* → ...*DAV4* → *DAV1* → *DAV2* → ...*DAV4*), both the LSB and the MSB of a single sample point from all eight channels have been transferred to the slave modules. This cycle is repeated 512 times, till all data has been transferred. The

control logic operation is detailed in the next section.

3.3.4 Control Logic

The data acquisition from the analog preprocessing subsystem and the transfer of stored data to the slave modules is coordinated by the master board control logic (schematics in Figure A.1). The micro-control required, though subtle, is not complex enough to warrant the use of a microcontroller. Two finite state machines implemented on three PALs (22v10s) were sufficient for our purposes. These PALs are clocked by a 1 MHz oscillator. PAL_AD has the FSM primarily responsible for reading conversion results from the AD7874s and writing them into memory. PAL_TR1 and PAL_TR2 together implement the FSM that coordinates the transfer of data from memory to the slave modules. Both FSMs must update the address counters and their buffers. In addition both FSMs must coordinate with each other to ensure that the correct memory address counter (*either* AD_COUNTER *or* TR_COUNTER) is addressing the RAM at any given time. That is, they must relinquish their shared resource, the SRAMs, whenever necessary, to ensure the integrity of the acquired data.

Much of the control requirements have already been made clear in the preceding sections. A discussion of the finite state machines and a listing of the PAL-generated control signals should completely explain the workings of the control logic. The PAL listings in Appendix C detail both the FSMs implemented. They also include a list of all PAL control signals. To explain the design and motivation for our particular FSM, a chronological listing of the events to which the control logic responds is given below. This will clarify the PAL code in Appendix C.

First, we consider the actions of the “AD FSM”:

1. The ADCs start a conversion in response to the \overline{CONVST} trigger pulse.
2. Upon the completion of a conversion, the ADCs assert their respective \overline{INT} signals. PAL_AD monitors the \overline{INT} s from both ADCs and, on seeing them asserted, prepares to begin ADC read operations.
3. As a first step, PAL_AD asserts the PAL_AD_BUSY signal, to tell the Transfer FSM that it needs the SRAMs. It then checks if a transfer to slave modules is in progress, by checking the PAL_TR_BUSY signal from PAL_TR2.

4. If a transfer is in progress, the AD FSM waits for the Transfer FSM. The latter sees the PAL_AD_BUSY signal asserted, and responds by finishing its current sample point transfer, going into a wait state, and deasserting PAL_TR_BUSY. It will resume the transfer once the AD FSM is done with its ADC read.
5. Once it sees PAL_TR_BUSY deasserted, the AD FSM proceeds with the ADC read. It repeats the following cycle 4 times:
 - (a) Assert \overline{CS} and \overline{RD} [both ADCs read simultaneously].
 - (b) Enable the LSB Buffers.
 - (c) Enable the SRAM \overline{WE} [both SRAMS are written to simultaneously].
 - (d) Deassert these signals to complete the write.
 - (e) Increment AD_Counter.
 - (f) Repeat the above steps for the MSB buffers to transfer the high byte to memory.
 - (g) Check if the three LSBs of AD_Counter are 000, i.e., if eight writes (of 4 sample points, one from each channel) have occurred.
 - (h) If not, repeat cycle.
6. The AD FSM deasserts PAL_AD_BUSY, and waits for another conversion. If the TR FSM was interrupted in the middle of a transfer by the AD FSM, it will now resume its operation.

Here are the operations performed by the TR FSM:

1. Once 4K of data (1K per channel) is in *each* SRAM, AD_Counter's MSB (RAM Address Bit 11) will have transitioned from 1 to 0. This transition triggers the TR FSM to start a transfer to the slave modules.
2. The TR FSM asserts PAL_TR_BUSY. If an ADC conversion is being written to memory, it waits for the AD FSM.
3. If the PAL_AD_BUSY is not asserted, the TR FSM asserts the Master-Slave handshaking signal *INT*, to interrupt all slave modules expecting data from the master. This line goes to the NMI (Non-Maskable Interrupt) line of the slave processors, so that after

a suitable wait (about 8 μ s) the master can be sure all slaves are listening. This way, one-sided handshaking is all that is required.

4. Both the `AD_Counter` and `TR_Counter` output buffers are controlled by `PAL_TR2`. When not in transfer phase, the `AD_Counter` buffers are enabled so that in the default case, RAM addressing is done by the `AD_Counter`. When the TR FSM enters the transfer phase, `AD_Counter` buffers are disabled and the `TR_Counter` buffers enabled.
5. The TR FSM repeats the following steps until all 4K bytes are shipped out from both SRAMs:
 - (a) RAM Output Enable (\overline{OE}) is asserted.
 - (b) `DAV1` is asserted. This latches the Channel 1 and Channel 5 data into the corresponding output registers. It also tells the listening slave modules that data is available.
 - (c) \overline{OE} is disabled; `TR_Counter` is incremented.
 - (d) \overline{OE} is enabled; `DAV2` is asserted; \overline{OE} is disabled; and again, `TR_Counter` is incremented
 - (e) `DAV1` may now be deasserted as enough time has passed for the slave modules to note data availability and respond to it. (See slave software listings in Appendix D.)
 - (f) The above steps are repeated for `DAV3` and `DAV4`. At proper intervals, `DAV2` and `DAV3` are deasserted.
 - (g) The LSBs have now been transferred (remember the order of stored data in the RAMs). Another round is initiated starting with `DAV1`. Note that `DAV4` will be deasserted early in this round.
 - (h) Finally, one sample point (two bytes) from each channel has been transferred. The TR FSM now checks if the AD FSM wants to write the results of a conversion to memory, i.e., if `PAL_AD_BUSY` is asserted. If so, it deasserts `PAL_TR_BUSY`, and goes into a wait state until `PAL_AD_BUSY` is deasserted.
 - (i) If not, TR FSM checks if the above cycle has been repeated 512 times, that is, if the MSB of `TR_Counter` has transitioned from 1 to 0. This MSB is tracked in the

following way: A Flip-Flop (74LS74) is used to latch the state of the TR_Counter MSB during the above cycle of transfers. At the end of the cycle this *old state* is compared to the *current state* of the MSB. If the 0→1 transition has not taken place, the above cycle is repeated, otherwise the transfer is complete.

6. After all 512 samples from all eight channels have been transmitted, PAL_TR_BUSY is deasserted, TR_Counter buffers are disabled, and AD_Counter buffers enabled. The TR FSM once again tracks AD_Counter's MSB to see when the next 512 point block will be ready for transfer.

Refer to Appendix C for a complete listing of PAL code and pinouts.

3.3.5 Sampling Rate Generator

The \overline{CONVST} input to the AD7874 needs to be pulsed regularly in order to obtain spectrally pure samples. This signal is provided by the Sampling Rate Generation Module (refer to Figure 3.1). This module comprises an EPROM, a programmable interval timer (8254), a PAL for micro-control, and a dip switch and resistor strip for frequency selection.

The 8254 is a general purpose multi-timing element, capable of being configured as an event counter, rate generator, square wave generator and real-time clock [16]. It is programmed by writing to its Control Word Register via its 8-bit bidirectional data port. The other three registers Counters 0, 1, and 2 are also initialized with the required count word. These four internal registers are selected by the address bits A1 and A0. In order to configure the 8254 as a square wave generator (which is the function needed for our purpose of sample rate generation), we must write the correct byte to the control word register. In order to set the frequency, we must write the LSB and MSB of the desired count into Counter 0 (the counter we choose to generate \overline{CONVST}).

The EPROM is needed to store all the various counts corresponding to the range of frequencies from which we can choose our sampling rate. The two LSBs (A0, A1) of the EPROM Address are driven by the PAL which can sequence through the three words stored at any particular offset. The next nine bits (A2–A10) are determined by a dip switch setting. This allows us to choose the correct EPROM address offset, and hence the count, to produce the desired sample rate. The expected range of sampling rates is 50Hz to 300Hz.

The Rate Generation PAL is responsible for the correct initialization of the 8254 by

loading the control data from the EPROM into the proper 8254 registers. The PAL goes through this sequence upon power up. In addition, a switch connected to a PAL input via a schmitt trigger allows the MLM user to reset the sampling rate by simply resetting the dip switch and pressing a button. The rate generator PAL Code is listed in Appendix C along with the PAL pinout.

3.3.6 Reset Circuitry

The reset circuitry is shown in Figure A.2. It consists of an RC configuration in parallel with the Reset switch, and a protection diode. The pulse generated is conditioned and buffered through two schmitt triggers in series. The output is a negatively-asserted signal, \overline{RESET} , that is fed to all the PALs on the master board. In addition, another signal, the positively-asserted $RESET$, is taken from the output of the first schmitt trigger and applied to the CLR input of the HC4040s. In the same schematics is shown a similar circuit which is used to provide the $LOAD_RATE$ signal to the rate generator PAL. Note that this allows the sample rate to be set at power up or be reset manually, as mentioned before.

3.3.7 PC Interface

While the primary function of the master board is to collect and transfer data to the slave modules, we can do more with the data acquired. Specifically, the data can be shipped to the host PC to allow, for instance, comparison with the transient event detection results produced by the slaves. This data would also allow transient templates to be collected during the MLM training phase (see section 5.2). For this purpose, the master board incorporates a PC interface section with its own data buffer, output registers, connectors, and control PALs. The schematics are included as Figure A.4.

The data buffer consists of two 32Kx8 SRAMs addressed by a single counter, the HC4040. Note that this address counter is a separate HC4040 in the PC interface section and not the $TR_Counter$ or the $AD_Counter$. It provides address lines A[03:14] to the memory ICs (the PC interface PALs provide A[0:2], as explained below). These RAMs each store 8 times as much data as each storage RAM. Each SRAM has an input buffer on its I/O lines which is the gateway between the output of the AD7874 buffers and the PC interface SRAMs. By disabling these buffers, the PC interface PALs can disconnect the SRAMs from the rest of the acquisition and storage circuitry, and can then ship data from

these RAMs to the PC. The I/O lines of the SRAMs also drive output registers (LS574s) whose outputs go to connectors linking the master board to the PC. A 16-bit word of data (a byte from each RAM) is thus presented to the PC via the connectors. These connectors also carry handshaking signals.

While the PC does not request data, the PALs keep the RAM input buffers enabled. Every time PAL_AD writes conversion data to the two storage RAMs, the data is simultaneously written into the PC RAMs. The PC RAM Counter is updated to keep pace with the AD_Counter. Using the RAM \overline{WE} and \overline{OE} lines, and the CLK_CNT from PAL_AD to control the PC Interface RAMs, and their counter, would not be correct. While this would synchronize the PC Interface Buffer with the Data Acquisition Section, it would not allow the PC Interface PALs to isolate the PC RAMs from the ADC section, when it wants to access the data there and send it to the PC. Hence, the synchronization is done through the PC PALs who monitor the RAM \overline{WE} line and other relevant PAL_AD signals to see when an ADC conversion result is being written to memory. Furthermore, not only do the PC PALs update the address counter, but in fact directly provide the three least significant address bits. This is necessary because 15 lines are needed to address the 62256 RAM IC, and the 4040 is a 12-bit *non-cascadable* counter. This scheme is less cumbersome than incorporating several 4-bit counters in the design. In a way it is also more *symmetric*: A/D conversion results are written in 8 byte blocks, so the PALs need update only their internal address counter during the AD data storing sequence and not worry about the external address counter. Once the memory writing is completed, the 4040 is incremented to record the acquisition of a new set of sampled data. Refer to the PC PAL code listing in Appendix C.

The PC PALs receive two communication signals PC_MODE and PC_FETCH from the host PC, and send a Data Valid signal, DAV2PC, to the PC. In addition, a fourth connection labelled PC_MISC, is provided to be used as an input to, or an output from, the master board. This hardware configuration allows for flexibility in designing the PC software for interfacing with the MLM master board. Several handshaking schemes are possible. The following sequence is the one implemented in the master board.

1. The PC asserts the PC_MODE signal.
2. The PC PALs responds by entering into *data transfer* mode, asserts PC_MISC, and waits for the PC to request data.

3. Upon seeing `PC_MISC`, the PC executes the following cycle 32K times:
 - The PC asserts `PC_FETCH`.
 - The PALs place the first data word on the output, increment the counter, and assert `DAV2PC`.
 - The PC reads in the data and deasserts `PC_FETCH`.
 - The PC PALs deassert `DAV2PC`.
4. Once the transfer is complete, the PC deasserts `PC_MODE`.
5. The PALs deassert `PC_MISC`.

3.3.8 Miscellaneous Hardware Components

In addition to the IC's and discrete components, the MLM master board includes hardware components discussed briefly below.

Jumpers

In order to provide flexibility in the choice of the final implementation, and leave software decisions to the end, the following jumpers were used on the master board:

- **J15:** Selects whether PC PAL2 takes in the global *CLK* signal or an output from PC PAL1 at its Clock Input, pin 1 (see Figure A.4).
- **J18:** Selects whether `PC_MISC` will be a PAL output or input (see Figure A.4).
- **J19:** Decides whether the *Gate0* input of the 8254 comes from the `CLK_GEN` PAL or is connected to V_{cc} (see Figure A.2).

Decoupling Capacitors

For roughly every two ICs on the board, a 0.1uF decoupling capacitor is placed on the board. These capacitors are placed as close as possible to the chips (within 0.030 inches) and usually at their front end (before pin 1). Decoupling capacitors are also used between $-12v$ and GND, and $+12v$ and GND, near the 7905 voltage regulator. A bigger tantalum capacitor is used near the power source, between GND and $+5v$.

Component	Quantity
A/D Converters	2
Memory ICs	5
Prog. Timer	1
PALs	6
Counters	3
Registers	10
Buffers	10
Misc. ICs	4
Total ICs	41
Discrete Comp.	47
Connectors	15
Total Comp.	103

Table 3.1: Master Board Component Listing

Headers linking Host PC to Slave Modules

Standard-size DIN (Dual-in-line) 20 pin headers are used in the master board to interface with the slave boards and the host PC. The lead spacing between pins is 0.100 inch. See schematics in Figure A.3 for the ordering of the signals on the connectors.

In addition, a DB9 connector and a 5x2 DIN header are provided on the master board (see Figure A.3). These are not connected to any components on the board and indeed, are *not* part of the master board design. They carry signals used for PC communication with the slave modules. Placing the DB9 connector on any one particular slave board would have been asymmetrical, and placing them on all slave modules would have been wasteful. These connectors on the master board are *relays* for the exclusive use of the slave modules.

3.4 Hardware Specifications

The MLM master board is a 12in. x 12in., 2-layer printed circuit board (PCB). The schematics and layout for the PCB were developed using the Personal Automated Design System (PADS). Refer to [11] and [12] for details of this CAD tool.

Table 3.1 lists the components that populate the master board. There are 103 components on the board. Of the 41 ICs, the ADCs are by far the most sensitive to power level changes, requiring the +5v and -5v inputs to remain within 5% of their expected value. This places strict limits on V_{cc} , delivered from the power supply, and on V_{ss} at the output

of the LM7905 voltage regulator:

$$4.75v \leq V_{cc} \leq 5.25v$$

$$-4.75v \geq V_{ss} \geq -5.25v$$

As the master board must transfer data over cable to several slave modules, its current, and therefore power consumption, is significant. Its supply current rating is nearly 2A under normal operating conditions, bringing its power consumption to about 10W. If the master board and one or more slave boards (their current requirement is comparable) are driven by the same supply, it must be ensured that the power supply is capable of delivering the requisite current to the boards. Otherwise V_{cc} may drop below the required 4.75v.

Master board schematics are given in Appendix A. PCB layout plots showing component placement are also included.

Chapter 4

The MLM Computational Units

This chapter explores in detail the function, design and implementation of the basic MLM Computational Unit, the **Slave Module**¹. We begin in Section 4.1 with the block-level design of a MLM slave module. In the context of this discussion and the treatment of the computational model of the MLM in Chapter 2, Section 4.2 explains how the slave processors perform their various functions. Finally, in Sections 4.3 and 4.4, details of the hardware design and implementation of the slave module and the slave board are given.

Schematics for the board, as well as final layout silkscreens, are included in Appendix B. They should be referred to throughout this chapter and, in particular, during the reading of Sections 4.3 and 4.4, where implementation details are laid out.

4.1 Design Overview of a Slave Module

The MLM contains several slave modules operating in parallel and communicating with one another. The slave module is the primary functional unit of the MLM. The slave module's design was governed by the following aims:

- Flexibility (in function assignment): A slave processor must be able to perform any task in the transient event detection algorithm. Not only must each processor be able to perform the computational tasks, it must also accommodate the different modes of master-slave and slave-slave communication that are needed for the various functions.

¹In this thesis the terms *slave module*, *processor module*, *slave*, *processor* and *slave processor* are used interchangeably to mean the basic computational unit of the MLM. The term *slave board* refers to the actual printed circuit board (PCB) which houses four slave modules.

- **Architectural Symmetry:** Flexibility of functionality must be achieved without custom hardware additions: A single slave module architecture must be developed that may be programmed to perform any step in the algorithm.
- **Scalability:** The architecture should be perfectly scalable so that an increase in computational power may be simply and robustly provided by adding on more slave modules to the multiprocessing system.

4.1.1 The Processing Engine

Figure 4.1 shows a block-level diagram of the architecture of a slave module. The centerpiece of the slave module is the Intel 80C196KC microcontroller [13]. Both Read Only Memory (ROM) and Random Access memory (RAM) are provided. The microcontrol of all data paths is performed by the Microcontrol PAL (see Appendix C) in conjunction with the 80C196KC. The controllers internal UART is used for inter-slave communication. An external UART was therefore needed to carryout communication with the PC. A parallel port, Port 1, of the microcontroller is used for accepting data from the master board. In the case of slaves working on time scales derived from the original data, data is accepted from the “data decomposing” slaves, also called *master-slaves*. A four-bit input port, the High Speed Input (HSI) Port and a four-bit output port, the High Speed Output (HSO) Port are used for inter-slave communication.

4.1.2 PC Interface

The Host PC communicates over a serial link with the slave boards. It accesses the slaves to perform the following functions:

1. Download program code into a slave’s RAM.
2. Download templates needed for transient event detection.
3. Retrieve results of event detection and collation from the collaters.
4. Issue a software reset to the processors.

As shown in Figure 4.1, a UART IC, the Intel 82510, is present in each slave module and carries out the serial communication with the PC. The support interface circuitry consists of

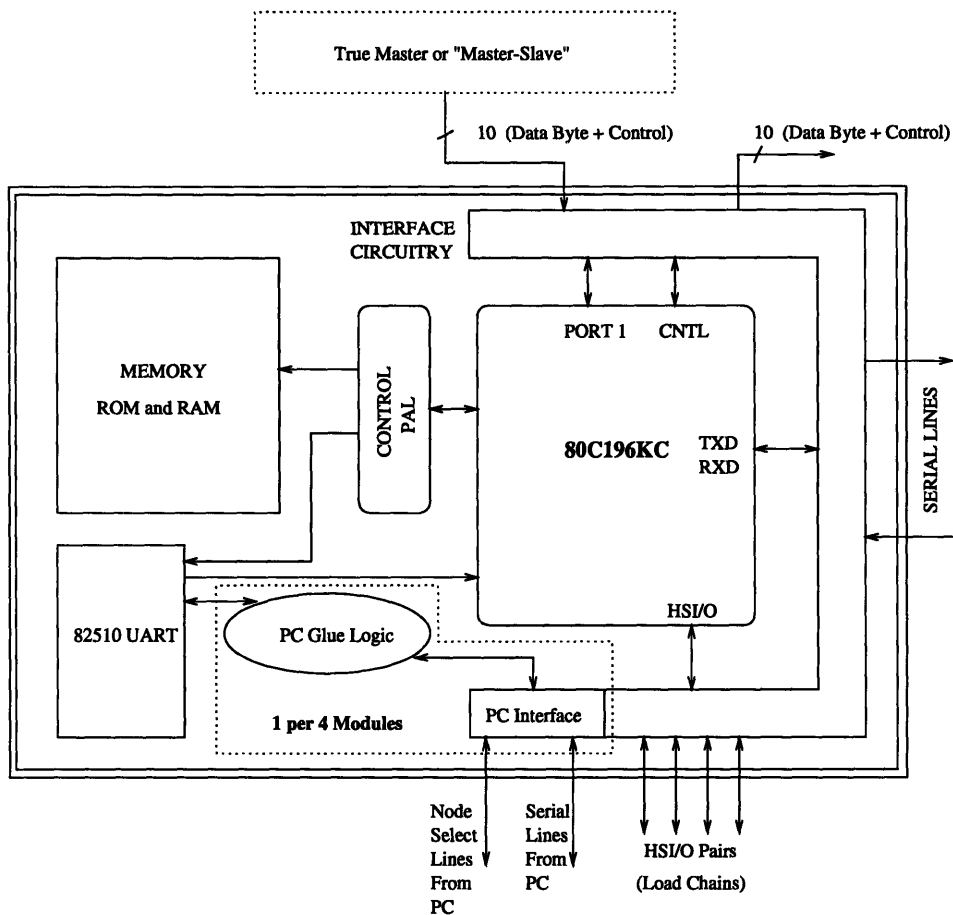


Figure 4.1: The MLM Slave Module

connectors, etc., to allow the routing of the RS-232 signals to each slave and allow on-board daisy-chaining of cables coming from the PC and going to all slaves.

As the architecture is highly scalable, an MLM prototype may consist of several slave boards, each housing four slave modules. Hardware support must be provided to enable the PC to select one processor at a time and address all communication to it. The implementation is simple: Each slave processor is assigned a unique 8-bit ID. The host PC sends the selected slave's ID byte via a PC I/O card (see Appendix F) to each slave board. The actual processor selection is achieved by special circuitry, called the "glue logic", on the slave boards. The glue logic accepts as input, the serial link to the PC as well as the serial lines of all four slaves. It compares the provided slave ID byte with DIP switch settings to determine if its board is selected and, if so, which processor is selected. It then connects the accessed slave processor's serial lines to the PC serial lines, allowing communication to

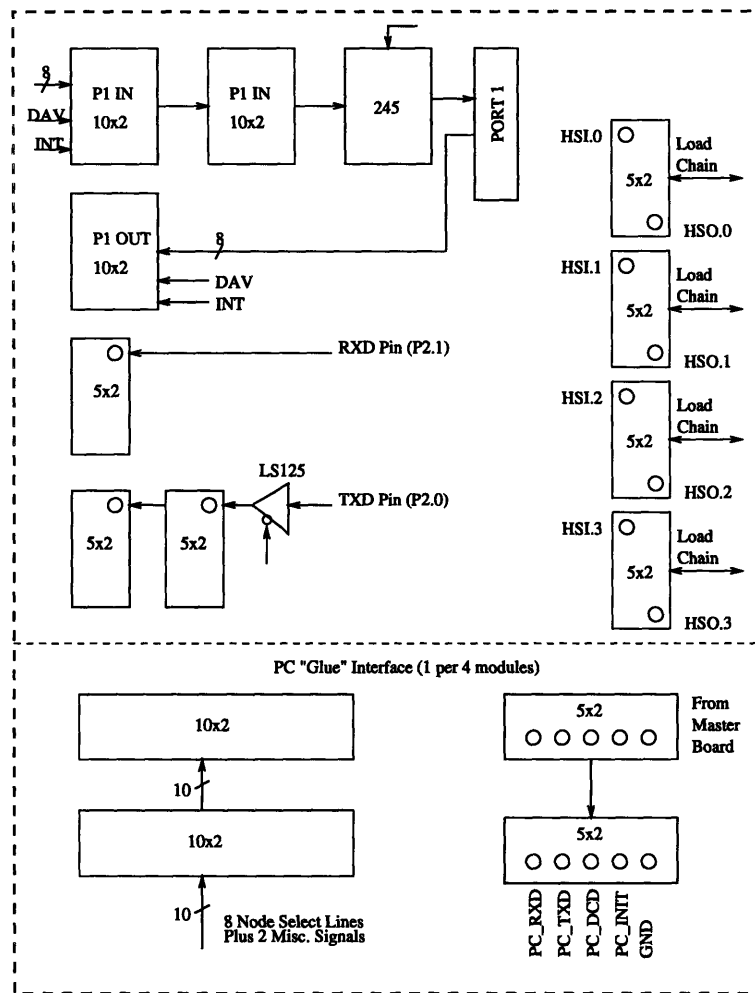


Figure 4.2: The Interconnection Circuitry for a Slave Module

proceed. Figure 4.2 shows the connectors needed to implement this scheme. To allow ease of scalability while maintaining architectural symmetry, the serial channel of the PC goes to a DB9 connector on the *master board*. From here it is distributed to all the slaves via ribbon cables. Placing a DB9 connector on a single slave board would have led to an asymmetrical design. Placing it on every slave board would have been redundant. Each slave board has, as part of its glue logic circuitry, two identical connectors to allow on-board daisy-chaining of the cables carrying the serial signals. The processor ID connectors are also duplicated on each board for this purpose.

4.1.3 Master Board Interface

The master board sends the acquired data to the slave processor via an 8 bit parallel link. The slaves receive the data on their Port 1. The interrupt signal from the master goes to each slave's Non-Maskable Interrupt (NMI) line. This means that regardless of the task a slave may be performing when the master decides to send it a shipment of data, it will be ready within 61 state cycles ($7.6\mu\text{s}$) to acquire data from the master. The transmission of each sample is controlled by the Data Available (*DAV*) signal of the master board, which the slaves accept as an input on a port 2 line.

Slave processors searching for transient events on time scales different from that of the originally sampled data, acquire input data from other slaves performing tree-structured decomposition. The elegance of the master interface circuitry and the data transfer protocol means that, for these slaves working on the new time scale, it does not matter whether their "master" is the master board or another slave. The communication asymmetry occurs only in the case of the data decomposing slaves. If these were configured as event detectors instead of decomposers, they would have used Port 1 strictly as an input port. Now, however, they must *output* data on Port 1 as well (ports 2, 3, and 4 are not available for this function). The simple solution implemented allows this flexibility without sacrificing symmetry. As shown in Figure 4.2, a buffer is placed between the input connectors carrying data from the master and port 1. When the decomposing slaves are ready with data to be sent to the slaves, the buffer is disabled. Port 1 is now used to output the decomposed data. Once the transfer is completed, the buffer is re-enabled and the link with the master board is re-established. Note also in Figure 4.2, the use of two connectors to receive the cable from the master board, allowing for daisy chaining of cables on-board.

4.1.4 Inter-Slave Communication

In order to implement the abstract model of the MLM presented in Chapter 2 over a distributed network of processors, one of the most important features is hardware support of inter-processor communication: as load v-sections are distributed over several processors, all processors associated with the identification of a load must be able to communicate their findings to one another. For this purpose the High Speed Input (HSI) Port, and the High Speed Output (HSO) port are used. Four 5x2 connectors are provided in each slave

module. Each connector is supplied an HSI line and the corresponding HSO line, pin 1 of the connector getting the HSI bit and pin 10 getting the HSO line. This bitwise pairing and signal placement allows the output signal from one slave to go to the input signal of the other slave, and vice versa, by a simple ribbon cable connection. This scheme allows two-way handshaking between the slaves. In the present implementation this handshaking was enough to tell slaves searching for the same load on a particular time scale whether one slave has found all its v-sections or not — the serial link was not used between every slave module. Note how this scheme is independent of whether a slave is configured as an event detector, a collater, or a data decomposer which also performs event detection.

The serial port is also used in interprocessor communication in conjunction with the HSI/O ports. In the current implementation of the MLM the serial link is used only to relay a load identification to collaters. However, if one were willing to bear the increased hardware and software complexity, the serial link may be used between event detector processors as well. Even in the serial port's simple use for transmitting results to the collater, consideration must be given to the fact that a collater's RECEIVE line accepts several processor's TRANSMIT signal. In order to avoid contention, the serial link is treated as a *shared bus* and the TRANSMIT line of each slave module is made to go through a tri-state buffer. Only when (via the HSI/O lines) a slave gets the go-ahead from the collater, can it take hold of the bus and transmit serial data.

4.2 Functional Overview of the Slave Module

The slave module can be configured to perform any one of the following functions:

- Transient event detection on one of several time-scales for one or more loads.
- Tree-structured decomposition and communication of the processed data to other units.
- Gathering of results from pattern recognizers, collating these results and relaying them to the PC.

Having understood the basic design of the MLM slave module, we now detail how these modules function as event detectors, data-stream decomposers, or collaters, and how the interprocessor communication works. Since our discussion here builds on the abstract model

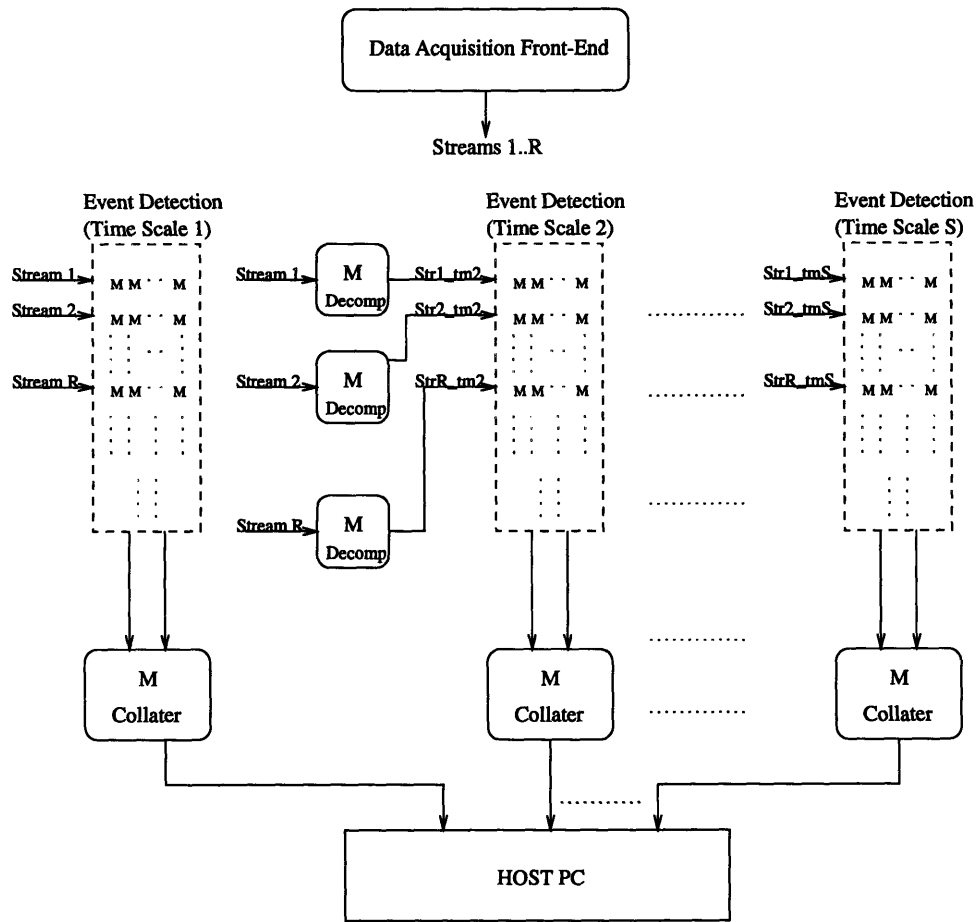


Figure 4.3: The MLM Model

of the MLM developed in Chapter 2, the diagram of our MLM model is reproduced in Figure 4.3 as reference for subsequent sections.

4.2.1 Event Detection

All slave modules periodically receive windows of data (in the present implementation, 512 sample points) from the master board (or a “master-slave”), comprising the input stream. Each slave module configured as a transient event detector has the templates of the v-sections that it is to detect, downloaded into its memory at startup time. Each slave may have v-sections belonging to the same load, or it may include v-sections belonging to more than one load if there is sufficient computation time available. Moreover, v-sections belonging to a single load may be distributed over several processors as most loads display

characteristic transients in several input streams (P, Q , 3P,..., etc.) and a slave processor can work on only a single input stream.

The pattern discrimination technique employed in the MLM is euclidean filtering. In this scheme, the aggregate of the point-wise absolute difference between two N-vectors is calculated. This is the *euclidean distance* between the two vectors. The euclidean distance between two N-point vectors, i and t , is:

$$\sum_{n=0}^{n=N-1} |i_n - t_n|$$

Hence an N-point v-section template t , would be matched against N-point sections or *subsequences* i , of the input stream according to the above formula. If the euclidean distance between t and i is within the error threshold, the v-section is said to have been positively identified i.e., a *v-section identification* is said to occur. In the MLM slaves, the input data is *ac-coupled* prior to filtering. The v-section templates are also ac-coupled. This make v-section identification possible even if a v-section occurs with a dc shift in the input stream. This is essential to our goal of detecting v-sections occurring in the quasi-static regions of another load’s transient, when two load transients overlap. The euclidean filtering of ac-coupled vectors that takes place in the slaves, may be represented as:

$$\sum_{n=0}^{n=N-1} |i_{n(ac)} - t_{n(ac)}|$$

As shown in Figure 4.3, a slave processor works on a particular time scale and inputs a single data stream from the master board (or a “master-slave”). It iterates over the input data, calculating the euclidean distance between a template and input subsequences. The procedure is repeated for each template that the slave is supposed to identify, and each v-section identification for a load is noted. Once all the v-sections for a load on a processor are identified, the processor announces this fact to the other processors identifying the rest of the v-sections on the given time scale for that load. A *load identification* on that time scale is said to occur when all the v-sections for that load have thus been identified. Load identification information is then passed to another slave processor, *the collater* (see Section 4.2.3), which collates the all event detection results on a given time scale. If a load has to be identified on more than one time scale, the collaters for each time scale will get the

identification results from the event detectors and pass on their results in turn to the host PC. The final verdict on whether the load has been identified will be made by the host PC.

As the set of v-sections associated with a load may be distributed over several processors, a procedure was devised for interprocessor communication which would allow v-section identifications to be translated into complete transient identification. Communication of v-section identification is done over the High-Speed Input/Output (HSI/O) port lines of the processors. The serial port is used to report a load identification to the collater (see Section 4.1.4). Connecting every processor associated with a load to every other processor would have been tedious and unnecessary. Instead, for each load detected by the MLM, all the processors identifying its v-sections on one time scale are *linearly* linked together via the HSI/O port lines to form a single chain of processors called a *load chain*. The processor at the head of this chain is designated as the “*first processor*” and the one at the tail is the “*last processor*”. Each processor on the *load chain* is aware as to its being the *first*, *last* or an *intermediate* processor on the chain.

Communication follows a **Generate and Propagate** scheme. If the *first processor* identifies all of its v-sections for a given load, it **generates** an “*identification message*” towards the next processor on the chain. This message is **propagated** by the *intermediate processors* upon detection of their v-section(s), until the *last processor* receives this message. Like all other processors on the chain, the *last processor* must also wait to identify its v-sections. If the *last processor* also identifies its v-section(s), a load identification is said to take place. When this happens, the *last processor* compiles a record comprising all relevant information (the transient location, the associated load, the event time, etc.). This is, as mentioned, then serially transmitted to a collater.

As an example, consider Figure 4.4, displaying seven processors M1, M2,...,M6. M1 through M5 are configured for transient event detection. M1 accepts as input real power, P. M2 and M3 work on reactive power, Q. M4 searches the third in-phase harmonic of current 3P. M5 works on the quadrature fifth harmonic, 5Q. Processor M6 is a collater for their time scale. Two loads, L1 and L2, are being monitored. L1 has one or more v-sections in P, Q and 3P. Its identification is the responsibility of M1, M2 and M4 and the load chain for load L1 is thus:

$$M1 \rightarrow M2 \rightarrow M4$$

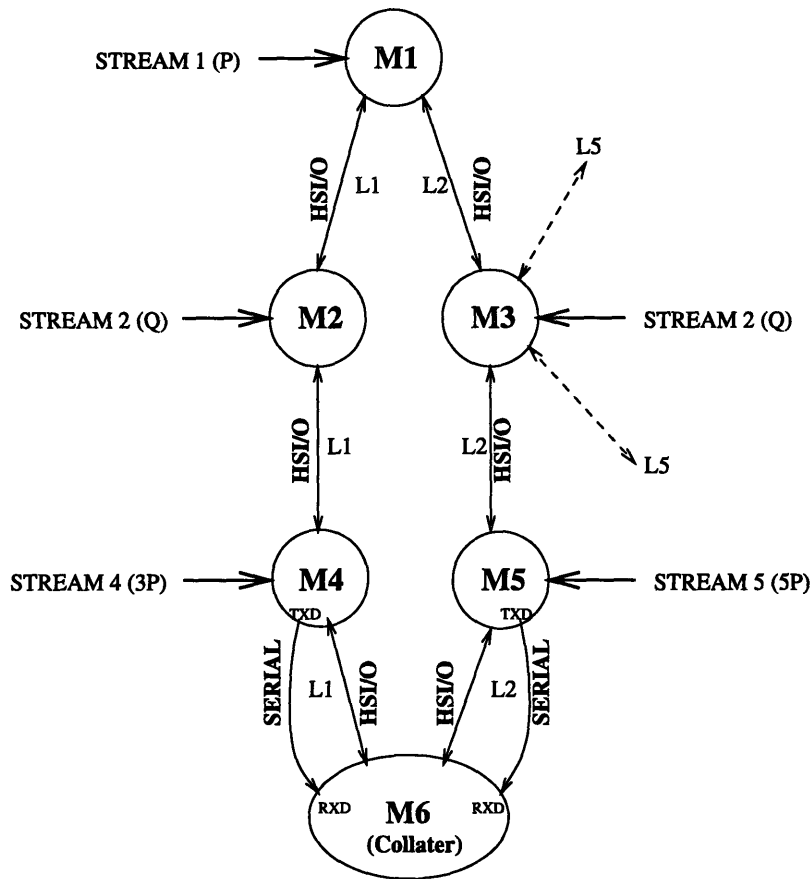


Figure 4.4: Load Chains in the MLM

Load L2 has one or more v-sections in P, Q and 5Q. The chain for load L2 is:

$$M1 \rightarrow M3 \rightarrow M5$$

M2, the *intermediate processor* for L1, waits for the *identification message* from M1. If no such message is received, M2 may not generate a hit message to M4 even if its v-section set is detected. Suppose M1 now detects its v-sections in stream P and sends an identification message to M2. If M2 has already identified all its v-sections in Q, this message would then be propagated to M4. M4, the *last processor* would then transmit a *load identification record* to the collater M6, if and only if, it too detects all its v-sections in its input stream, 3P. Note that any processor may, like M1 or M3 (which is part of the unshown load chain for load 5), be a part of more than one load chain. The transmission of results in the chain for load L2 follows a similar sequence.

4.2.2 Tree-Structured Decomposition

As explained in Chapters 1 and 2, the transient event detector in [1] allows for v-section detection over multiple time scales. To implement this function in the MLM, certain slave modules perform the function of time scaling the input data-stream and transmitting the downsized data block to the slave processors working on that new time scale. In the MLM, one processor is dedicated to downsampling input data to each of the required time scales. Thus if we need two input streams (real and reactive power envelopes, P and Q, for example) to be searched on three time scales, we need two decompositions per stream. Hence, we would need four processors dedicated to tree-structured decomposition. As given in Figure 4.3, in response to an input data block, each *decomposer slave processor*, runs the tree-structured decomposition algorithm on the data, which to a first approximation is digital low pass filtering followed by decimation in time (see [1] for details of this algorithm). The resulting block is then transmitted to the *pattern recognizer* slaves working on this coarser time scales. This transmission is done using the exact protocol of the regular master-to-slave transmission. The *decomposer processors* thus perform the decomposition as well as set up the slaves to perform event detection on the resulting time scale.

As tree-structured decomposition is a less computation-intensive algorithm than pattern recognition via euclidean filtering, the *decomposer* slave modules may also have time to perform pattern recognition and be part of a load chain.

4.2.3 Result Collation

Once all the processors on a load chain have detected their set of v-sections, a load identification record is compiled by the last processor on the chain and passed on to the *collater module*. There is one collater per time scale. That is, the *last* processors on all the load chains for a time scale go to the same collater. The primary function of the collater is to perform intra-scale lock-out on identified v-section and relay all genuine load identifications on its time scale to the host PC. Communication between a collater and all the *last* processors is on a round-robin basis. The collater polls each processor to see if a load identification record awaits it. If so, the collater grants the processor access to its serial port and reads in the record. Serial communication was chosen for this data path as the amount of data transmitted is on the order of a few bytes, making it unnecessary to dedicate a parallel port

for this purpose. Handshaking for this data transmission is achieved using the High Speed Input Output (HSI/O) port of the processors.

Once the collater reads in the data, it performs the lock-out procedure: It has in its memory all the templates of the v-sections that comprise its associated loads, and if multiple load hits are detected, it determines whether the v-section from a more complex v-section set coincides with a v-section for another load with a simpler v-section set. If this is the case, the load with the simpler v-section set is not considered identified. The collater then tabulates a record in memory of the events detected and asserts a flag which indicates to the host PC that new data is available.

The host PC also communicates with the collaters on a round-robin basis, checking periodically for the assertion of a flag by the collater, reporting one or more positive identifications. Since each collater works on a single time scale, they cannot independently perform inter-scale lockout. The final steps (inter-scale lockout and result display/storage) in the multiscale event detection algorithm are assigned to the host PC which has the event detection results on *all* the time scales available to it.

4.3 Slave Module Implementation Details

Design details of slave module are now set forth. It may be helpful to refer to the schematics in Appendix B while reviewing this section.

4.3.1 Memory Section

The slave memory comprises two 32Kx8 ROMs and two 32Kx8 RAMs (refer to Figure B.1). The ROMs contain the RISM (Reduced Instruction Set Monitor) code. This is the code that the host PC expects to be installed in the *Target* (in our case, the slave modules), for the PC to communicate properly with the *Target*. The RISM consists of about 300 bytes of 80C196KC code, and provides primitive operations. Software running on the host PC uses the RISM commands to provide a complete user interface to the slave modules. A summary of the structure of RISM is given in Appendix E. To learn more about the structure and content of the RISM refer to [14]. Using the RISM routines, target software can be loaded into the RAMs. In addition to program code, the RAMs are used to store pattern templates and input data arrays, etc.

Both types of memory are in a 16-bit configuration with one IC for the *High Byte* and one for the *Low Byte* of data. The 80C196KC signals \overline{WRH} and \overline{WRL} are used for the high and low data RAMs respectively. The \overline{RD} signal goes to all four memory chips. The \overline{CS} signals come from the Microcontrol PAL. This PAL uses the memory address accessed and the address bus' LSB to determine the chip to be selected. A jumper for pin A14 of the SRAMs accommodates the use of either a 16Kx8 or a 32Kx8 RAM capacity (refer to Figure B.1).

4.3.2 Microprocessing Unit and Microcontrol PAL

The slave module's computational heart is the Intel 80C196KC, a 16-bit, CMOS microcontroller [13]. The 80C196KC is designed to handle high-speed calculations and fast I/O operations, making it a good choice for the MLM. It has 16 multiplexed Address and Data lines for interfacing with external circuitry, including program memory, RAM, and the external UART (the I82510). It provides two 8-bit digital I/O ports, High-Speed Input/Output lines and a Full-Duplex Serial Interface for processor-to-processor communication. At 16Mhz, it is sufficient for our computational purposes and is upward compatible with the 80C196KD which operates at 20MHz. 1K of on-chip RAM is useful in speeding through number-crunching on small arrays. Together with the microcontrol PAL (a 22v10), the 80C196KC controls all the data paths of the slave module (refer to Figure B.2).

The operation of the 80C196KC depends on the initial contents of the Chip Configuration Register (CCR). Among other things, this 8-bit register controls the way the microcontroller interfaces with external memory, allowing for a great deal of choice in the implementation of an 80C196-based system.

The most important feature of the external memory interface is bus width. For memory accesses the bus width is 16 bits. The CCR is initialized so that the microcontroller operates in the "Write Strobe Mode". This mode eliminates the need to externally decode high- and low-byte writes to the external RAM. The 80C196KC generates \overline{WRL} and \overline{WRH} instead of \overline{WR} and \overline{BHE} as in the "Standard Bus Mode" (Refer to [13] for further details of interfacing with external memory). For accessing the I82510 UART, an 8 bit data path is necessary. To allow dynamic setting of the bus width (to either 16 bits for memory access or 8 bits for UART reads and writes) the CCR must be configured to allow the input pin *BUSWIDTH* to control the width of the data bus. *BUSWIDTH* must be held high

during a 16-bit data transfer (program code, RAM access) and low during an 8-bit transfer (UART access).

Another important consideration is the time the processor must wait for slow memory devices to complete their actions. The internal control circuitry of the 80C196 allows the *READY* signal to be pulled low until an external device completes an operation. While *READY* is low, the Bus Controller in the 80C196 inserts wait states into the bus cycle. The number of wait states generated is specified by the contents of the CCR. For our purposes, the CCR has been configured to limit the number of wait states to 3. This wait feature is used whenever the UART is accessed. As it has a long access time (about 200ns), the microcontrol PAL pulls the *READY* line low while the UART is chip-selected.

The microcontrol PAL code is listed in Appendix C. It decodes the address bus to generate the Chip Selectors for the RAM, the ROM and the UART. In addition, it controls the *READY* and the *BUSWIDTH* lines of the microcontroller. Finally, it stores the current state of the RISM (“user” mode or not “not user” mode, as described in Section 5.2), in an output bit, MAP, which determines whether the program code is executed from the ROM or the RAM (refer to Chapter 5 and Appendix C).

Some discrete logic is also used in the implementation of the data path. Two octal latches (74LS573) are used to externally latch the Address. They are clocked by the Address Latch Enable (*ALE*) output of the 80C196KC. A STretched ALE (*STALE*) signal is generated using a Flip-Flop clocked by the *ALE*. This is used by the microcontrol PAL to generate the MAP bit. *STALE* is deasserted (i.e., the flip-flop is reset) by the output of an LS08.

4.3.3 PC Interface

Each module communicates with the PC via the Intel I82510, a CHMOS Asynchronous Serial Controller (see Figure B.3). The I82510 has a Bus Interface Unit, a Serial Module, a Timing Block, and a Modem Interface Module. The former two units are the ones chiefly utilized in the slave module.

The I82510 has a simple demultiplexed bus interface which consists of a tri-stated 8-bit bi-directional data bus and a 3-bit address bus. An Interrupt line, along with Read, Write and Chip Select pins are the remaining signals used to interface with the CPU. The I82510 is programmed through its registers which are divided into four banks — 35 registers in all. Only one bank is accessible at a time. The switching is done by changing the contents of the

bank pointer. Refer to [15] for detailed specifications of the I82510 architecture, registers, and programming.

The 80C196KC \overline{RD} line goes to the UART \overline{RD} input. The \overline{WRH} line goes to the UART \overline{WR} [Note that \overline{WRL} could also have been used, since in the 8-bit bus mode both signals are asserted for all writes]. The \overline{CS} comes from the microcontrol PAL. The Interrupt output, *INT* of the I82510 goes to either the Non-maskable Interrupt (NMI) or another external interrupt source of the microcontroller, depending on the setting of jumpers JP3 and JP9 (see Section 4.4.2). The positively-asserted *RESET* signal is applied to the *RESET* pin, while \overline{RESET} is applied to the \overline{RTS} input via a diode. During hardware reset, the \overline{RTS} pin acts as an input and is used to determine the System Clock Mode. The \overline{RTS} pin is driven low during reset, so as to configure the UART to generate its clock internally using a crystal oscillator, instead of accepting an external clock signal. The *TXD* and *RXD* inputs are connected, via the Glue Logic, to the corresponding signals from the host PC. Finally, the $\overline{DCD}/\text{ICLK}/\text{OUT1}$ pin is configured as an output and used by the microcontroller to indicate to the host PC whether the slave is executing code or awaiting PC command.

4.3.4 Interconnection Details

Figure 4.5 shows the connectors along with the connector numbers (**Jxx**) as they appear in the slave board schematics and layout, and illustrates the listing of the connectors that follows. The exact pinouts for all these connectors are given in the schematics in Appendix B.

- **J2**: The power connector for the slave board, carrying, in order: +5v, GND, +12v, -12v. Note that while inlets are provided for +12v and -12v, they are not used by any on the ICs on board. A 4-pin molex connector was chosen for compatibility with the master board, where the ADCs do require these signals. J2 is not shown in Figure 4.5 but is present in Figure B.5 in Appendix B.
- **J3, J4**: These connector accept the 8-bit processor ID data from the host PC.
- **J5, J6**: These connectors interface with the master board to allow the transfer of blocks of acquired data.
- **J7**: This is the output connector used by the decomposer slaves to send time-scaled

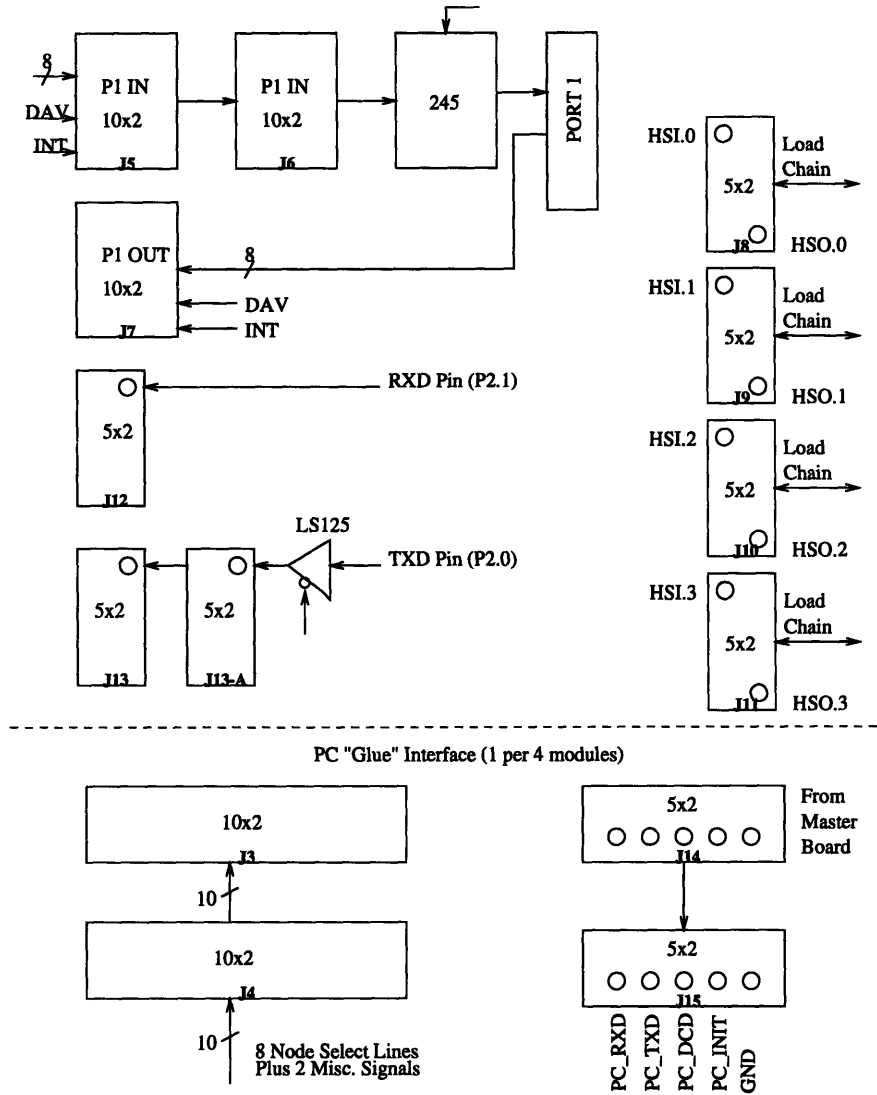


Figure 4.5: Slave Board Connectors

data to event detector slave.

- **J8, J9, J10, J11:** These are the HSI/O port connectors used in the implementation of the load chains. Each carries an HSI line and the corresponding HSO line.
- **J12:** This provides the input for the *RXD* line of the microcontroller.
- **J13, J13-A:** This is the output connector for the *TXD* line of the microcontroller. J13-A was a second connector added in the protospace to allow daisy chaining of the *TXD* lines of the processors communicating with the same collater.
- **J14, J15:** These connectors receive the serial communication signals from the host

PC.

4.3.5 Reset Circuitry

The Reset Circuitry is shown at the top of Figure B.2. The \overline{RESPIN} pulse is generated by an RC configuration for power-up reset, in parallel with a reset switch Wire-ORed with the \overline{INIT} signal from the HOST PC. This is in turn Wire-ORed with the $RESET$ pin of the 80C196KC. The total sources for slave module reset are:

- Reset Switch.
- Power Up.
- PC-issued reset through negation of \overline{INIT} signal.
- Internal 80196KC reset (via assembly language Reset Instruction, **RST**).

Finally, $RESET$ and \overline{RESET} are generated for the UART by means of a flip-flop (74LS74) clocked by " $\overline{WRL.WRH}$ " and cleared by the $RESET$ output of the 80C196KC.

4.4 The MLM Slave Board

Having discussed the MLM slave module's design and function, we now address the implementation of the slave board, the physical circuit board that houses the slave modules.

The MLM slave board is a 12in. x 12in. printed circuit board. As was the case for the master board, the schematics and layout for the slave board PCB were also developed using the Personal Automated Design System (PADS). Refer to [22] and [23] for details of this CAD tool. The slave board accomodates four slave modules, PC interface "glue logic", and some prototyping area. The slave modules have been detailed in the previous section. What remains is a treatment of the "glue logic" on board, which is responsible for interfacing the PC and the slave modules.

4.4.1 PC Interface Glue Logic

The host PC needs to download program code and data into each slave module. The PC also needs to retrieve results from the collaters. This communication is done serially as it is not a time-critical task. Since there is only one RXD line on the PC connected to the

*TXD*s of several slaves, and one PC *TXD* line transmitting to the *RXD* inputs of several slaves, the PC must be able to select one slave at a time. This capability is given by the Glue Logic on board each slave PCB.

Two PALs implement the PC interface. The “Comparator” PAL (see Appendix C) receives the six Most Significant Bits (MSBs) of the processor ID byte and compares these to the on-board DIP switch setting. If there is a match, one of the processors on the slave board is selected. The “Comparator” PAL asserts a *SELECT* signal to the “Relay” PAL which then makes the serial connection between the PC serial lines and a slave’s UART. The “Relay” PAL acts as a multiplexer in one direction and a demultiplexer in another: It demultiplexes the PC *TXD/RXD* lines to 1 of 4 slave *TXD/RXD* lines. At the same time it does a 4-to-1 multiplexing of the four sets of slave serial lines to the single set of PC serial lines. Upon being told by the “Comparator” PAL that the slave board is selected, the “Relay” PAL determines from the two LSBs of the processor ID which processor is selected. It then demultiplexes the PC *TXD* and *RXD* lines so that they are connected to the selected processor’s serial lines. Thus, when a module is selected, its UART is connected directly to the PC’s serial port. Other processing modules can have no effect on this link.

As the PC delivers its serial lines via RS-232 cables, a tranceiver IC is used to convert the PC signal voltages to TTL levels. The component used is the MAX235, an RS-232 line driver/receiver with on-board charge pump voltage converters. This means that the MAX235 can generate the needed $\pm 12\text{v}$ from 5v — no $\pm 12\text{v}$ power signals need to be provided. Its most useful feature, however, for the PC interface, is a *shutdown mode* in which the slave *TXD* lines are disconnected (put in high-impedance) from the PC *RXD* line. If a slave board is not selected, its “Comparator” PAL places the tranceiver in shutdown mode. This ensures that if a board is not selected, none of its processors can force the PC *RXD* line high or low.

4.4.2 Jumper Selections

Several jumpers are used in the slave board to allow flexibility in the configuration of the hardware. The jumpers in a slave module and the glue logic circuitry, and their selections are described below.

- JP1: Placed next to the microcontrol PAL, this decides which pin of the PAL provides the 80C196KC *READY* signal. In the default position (1-2), the I82510 \overline{CS} and the

READY signal are provided by the same PAL pin.

- JP2: Located in the glue logic circuitry, this jumper determines whether the glue logic PALs are reset by the \overline{RESET} line or the \overline{RESPIN} line. The default setting (2–3) chooses the \overline{RESPIN} line.
- JP3, JP9: Placed next to the I82510, these select whether the *INT* (interrupt) output of the I82510 goes to the NMI line, the P2-2 external interrupt pin or the P0-7 interrupt pin of the 80C196. The default setting (2–3 for both jumpers) chooses the P2-2 (the JP3 setting is actually irrelevant if JP9 is in the 2–3 position).
- JP4: This jumper selects the 80C196 input source for the Interrupt signal from the master, choosing between NMI and P2-2. The interrupt source selected for the master board interrupt by this jumper and the source for the I82510 interrupt (dealing with PC communication, selected by JP3/JP9) *must be different*. Otherwise the slave module will not be able to communicate with the PC or the master board. The default setting is 1–2 which selects the microcontroller’s NMI pin.
- JP5: This selects the 80C196 pin to which the *DAV* signal from the master will go. The current setting (1–2) connects *DAV* to input pin P2-3.
- JP6: The jumper selects the output pin of the 80C196 that will provide the interrupt signal when the slave is configured to perform tree-structured decomposition and needs to transfer time-scaled data to other slaves. The default setting (2–3) selects HSO-1 over P2-5.
- JP7: The jumper selects the output pin of the 80C196 that will provide the *DAV* signal when the slave is configured to perform tree-structured decomposition and needs to transfer time-scaled to other slaves. The default setting (1–2) selects HSO-0 over P2-7.
- JP8: This determines whether the \overline{OE} input of the buffer in front of Port 1 is tied to ground or controlled by P2-6. The present setting (2–3) uses pin P2-6.
- JP10: This jumper allows the 80C196 *TXD* line to either go directly to the output connector, J-13, or through a tristate buffer. At present the jumper is in position 2–3 so that the tri-stated *TXD* line is provided to the output connector.

- JP11, J12: These jumpers determine whether the \overline{OE} input of the tristate buffer for the *TXD* line is controlled by P2-7, HSO-0 or HSO-1. The present setting (1-2) for both jumpers (although the J12 setting does not matter once J11 is in the 1-2 position) chooses pin P2-7.
- JP13: This jumper allows the usage of either 32Kx8 RAMs or 16Kx8 RAMs. The default setting is (1-2) and the default size is 32Kx8 (256K).

4.4.3 Component Layout

The PCB Layout included as Figure B.6 in Appendix B should be referred to throughout this section.

The slave board has the four slave modules forming four symmetric quadrants on the board. On the lower side, from left to right are modules 1 and 2. The upper side has modules 3 and 4 (from left to right). In the middle of the board is the glue logic (left center) and some protospacing area (right center). The protospace was included to allow any additions/changes that may need to be made to the board during the test phase. It turned out to be extremely useful when the additional connector J13-A, in Figure 4.5. was added for the on-board daisy chaining of the *TXD* lines of the microcontrollers (see Section 4.3.4).

Along the left edge of each module are the four memory ICs. To their right, and near the top is the external UART, the I82510. To the right of this module are the discrete components and SSI logic making up the Reset Circuitry, as well as an AND gate (LS08) used in the generation of the *STALE* signal. Below this part of the board is the 80C196KC in a 68 pin PLCC package, with a pair of latches (74LS573) and the microcontrol PAL to its left. To the right of the controller and below it, are arranged the connectors for inter-slave and master-slave communication.

Component	Quantity
Microcontrollers	4
Memory ICs	16
UARTs	4
PALs	6
Misc ICs	30
Total ICs	60
Discrete Comp.	174
Connectors	45
Total Comp.	279

Table 4.1: Slave Board Component Listing

4.4.4 Hardware Specifications

Table 4.1 lists the components that populate the slave board. There are 279 components on the board, including 60 ICs. None of the ICs in the slave board are as sensitive to changes in V_{cc} as the AD7874 used in the master board. Hence, as long as TTL levels are maintained, no further restrictions apply. Under normal operating conditions, an I_{cc} of about 1.5A is required.

Chapter 5

Software Implementation of the MLM

Having covered the hardware design and implementation of the MLM, we turn to the software implementation of the transient event detection algorithm on the slave modules. The design and development of the Host PC Interface software is also included.

5.1 Software for Slave Processors

The 80C196 C source code for Transient Detection, Tree-Structured Decomposition and Result Collation is given in Appendix D. The salient features of the software design of each operation are discussed below.

5.1.1 Acquisition of Input Data

A slave module performing a transient pattern search is periodically interrupted on its NMI line by either the master board or a slave module performing tree-structured decomposition. A block of 512 input data points is then transferred to the slave processor. The following NMI interrupt servicing routine (see [17]) is responsible for accepting the data blocks:

```

void nmi_master_int(void)
{
label1:
while ((ioport2 & 0x08) != 0x08); /* wait for DAV = p2.3 */
input[counter] = (ioport1 & 0x3f); /***** 6 bits *****/
while ((ioport2 & 0x08) == 0x08); /* wait for /DAV */
counter++;
if (counter < INPUT_SIZE)
goto label1; /* repeat until 512 points received */

new_acq = 1; /* flag to indicate arrival of new data*/
no_of_acq++; /* no. of acquisitions up till now */
counter = 0;
}

```

5.1.2 Pattern Search

The main operation in the TED algorithm is searching the input stream for patterns stored as templates in memory. Currently, a single processor can search for v-sections belonging to three loads. The event detector code (given in Appendix D) may be modified to allow upto four loads per processor. Beyond that number, the limitation will come from the restricted fan-in/fan-out of the load chain implementation (see Section 5.1.3). Of course if the fan-in/fan-out restriction was not present, more loads could be handled by each slave module. Recall from Section 2.2 that 10 to 25 v-sections (average size of about 20 points) can be searched by a processor in the time available between consecutive input data arrivals. A processor can, therefore, easily handle more than four loads. Depending on the number of v-sections in each load's transient, a processor could potentially monitor v-sections for 10–15 loads.

The backbone of the software implementation for the v-section pattern search was the load structure [18] defined as follows:

```

struct load{
unsigned char used; /* load used? */
unsigned char mach_no; /*load id */

```

```

unsigned char prev_hs; /*For First Proc on load chain, no prev_hs..*/
unsigned char next_hs; /* ..and Prev_got must be set to one for it */
unsigned char no_vsec; /* no of vsections associated with load */
volatile short *vs1_add; /* address of 1st vsec */
unsigned short vs1_size; /* size */
volatile short *vs2_add; /* address of 2nd vsec */
unsigned short vs2_size; /* size */
volatile short *vs3_add; /* address of 3rd vsec */
unsigned short vs3_size; /* size */
volatile short *vs4_add; /* address of 4th vsec */
unsigned short vs4_size; /* size */
volatile short *vs5_add; /* address of 5th vsec */
unsigned short vs5_size; /* size */
unsigned char first_got; /* is this the first processor on chain? */
unsigned char last_got; /* is this the last processor on chain? */
unsigned char prev_got; /* Did prev. proc. send the ident. message */
unsigned short prev_time; /* acq_time for prev_got */
unsigned char vs_hits; /* no. of vsecs. identified so far */
unsigned short vs0_loc; /*offset from begin. of inp block for 1st vs*/
unsigned short acq0_time; /* acq_time for first vsec */
unsigned short vs_loc; /*offset from begin. of inp. block for last vs*/
unsigned short thresh; /* error threshold */
short range_hi; /*range within which first and last vsec should be*/
short range_lo; /*range within which first and last vsec should be*/
}

```

Each load whose v-sections are to be recognized is characterized by a number ID, mach_no. The time scale of operation is defined as a constant at the beginning of the program for each processor (see listing in Appendix D). The number of v-sections to be searched and the size and memory address of each v-section template for a load, are stored in its load structure. The variable prev_hs identifies the HSI/O pair on which this processor communicates with the previous processor on the load chain. Similarly, next_hs identifies the HSI/O line on which the processor communicates with the next processor

on the load chain. The flag `first_got` is set to 1 if the processor is the first processor on the load chain, and `last_got` is set to 1 if the processor is the last processor on the load chain. The variable `prev_got` is set if the previous processor on the chain has sent a load identification signal, and `prev_time` stores the “time” at which the `prev_got` was set. The number of v-sections found so far are recorded in `vs_hits`. The other elements of the load structure are used to perform a validity check once all v-sections have been found. For instance, the v-sections must be identified in order, the first and last v-sections must occur within a certain spatial range, and the identification must take place within a set time of receiving the previous processor’s identification message. Finally, an error threshold is assigned to each load in the variable `thresh`.

The time base for the MLM is the *number of data transfers that have occurred so far*. Thus, if the master board has transmitted 210 blocks of data, the system time is 210. If at this “moment” (i.e. during this data acquisition cycle) a load is identified, the identification time will be set at 210. Each slave has two global variables that keep track of time. The variable, `no_of_acq`, counts the number of data acquisitions (time ticks) that have occurred since the system was started up. The variable `acq_time` is set to the value in `no_of_acq` when a load identification occurs. It is passed on to the collater (and then to the Host PC) as the time at which the load was identified.

The steps in v-section search for multiple loads on a processor are as follows:

1. Wait for a new acquisition.
2. Once a new data block arrives, do the following for each load that is monitored:
 - (a) Search for first v-section of the load. If the first v-section is not found, check `vs_hits` to see if the first v-section was identified on a previous acquisition cycle. If this is also not true, end search for v-sections.
 - (b) If the first v-section is found in this cycle or is shown as found on an earlier cycle, check if there are more v-sections to be searched. If not, go to the next step. Otherwise repeat steps (a) and (b) for the next v-section.
 - (c) If all v-sections have been identified:
 - i. Check to see if `prev_got` is set. If not, then wait for `prev_got` to be asserted.
 - ii. If the previous processor *has* sent the identification message, perform validity check on the locations of the v-sections and the time difference between

the previous identification message being received and the v-section being identified. If the identification is invalid then reset the variable `prev_got` and `vs_hits` and start over.

- iii. If the identification is valid, send an identification message to the next processor on the chain. If the current processor is the last processor on the chain, compile an identification record and serially transmit it to the collater. Also reset the variables `prev_got` and `vs_hits`.

3. Go back to step 1.

Since the code is interrupt-driven, the processor does not stall while waiting for an event (such as the arrival of new data or of an identification message). In the above algorithm, v-sections are identified through *euclidean filtering*. The euclidean distance between two N-point vectors, i and t , is computed according to the formula:

$$\sum_{n=0}^{n=N-1} |i_n - t_n|$$

where t is an N-point v-section template and i represents N-point sections or *subsequences* of the input stream. If the distance is found to be less than the threshold for the load, the v-section is considered identified. As explained in Section 4.2.1, the input data is *ac-coupled* prior to filtering. The v-section templates are also ac-coupled. This make v-section identification possible even if a v-section occurs in the input stream, with a dc shift.

5.1.3 Software Implementation of Load Chains

Each event detector slave uses HSI/O lines to implement the load chains. Each processor allocates, for each load used, an HSI/O pair for communication with the previous processor on the chain. This is specified by the `prev_hs` element in the `load` structure. The element, `next_hs`, specifies the HSI/O pair used to propagate the identification message to the next processor in the chain. As a total of four HSI/O pairs are available on the 80C196KC, the total fan-in+fan-out is restricted to four. Each process has a servicing routine for the HSI Data Available Interrupt. If the HSI line corresponding to the `prev_got` line of a used load is set to 1 by the previous processor, an HSI Data Available Interrupt occurs. The servicing routine checks the HSI input on which the interrupting event occurred. If this

matches the `prev_hs` line of a used load, the corresponding HSO line is used to complete the handshake. At the same time, `prev_got` is set to 1 and the current acquisition cycle number is recorded in `prev_time`. If the processor is itself ready to propagate the identification message received, it will set the HSO bit specified by `next_hs`, to signal the next processor. If the load's `last_got` is set, then the next processor is a collater. A record consisting of the processor ID, load ID, time scale, acquisition time and v-section location (for intra-scale lockout) is compiled and transmitted to the collater.

5.1.4 Template Management

Templates for v-sections are placed at specific memory locations using the `#pragma` compiler directive [17]. These memory locations are listed below:

```
#pragma locate (template1 = 0x3600) /* 8 BIG TEMPS., 4 SMALL ONES */
#pragma locate (template2 = 0x3800)
#pragma locate (template3 = 0x3A00)
#pragma locate (template4 = 0x3C00)
#pragma locate (template5 = 0x3E00)
#pragma locate (template6 = 0x4000)
#pragma locate (template7 = 0x4200)
#pragma locate (template8 = 0x4400)
#pragma locate (template9 = 0x4600)
#pragma locate (template10 = 0x4700)
#pragma locate (template11 = 0x4800)
#pragma locate (template12 = 0x4900)
```

A total of 12 templates are allowed at present per processor. Eight of these may be 512 points long while the other four may only be 256 points long. Each load may have up to a maximum of five templates. The function `load_init()` initializes the v-section addresses and sizes for each load. See the listing of `p0_e1p.c` in Appendix D for an example of v-section initialization.

An important feature of the slave processor software is the template acquisition mode. The global variable "mode" is placed at memory location 0x3504. If this is set to 1 by the host PC, no pattern search is performed upon a new block acquisition. Instead the

subroutine `t_mode()` is called to determine if a change of mean occurred in the input data. If this is true, the processor enters an endless loop in the `t-mode()` subroutine from which it breaks out only if the “mode” variable is reset. Details of the template acquisition mode from the perspective of the user on the host PC are given in Section 5.2. Note that this feature is also present in “decomposer” slaves. Here, in template acquisition mode, tree-structured decomposition is performed and the scaled data is distributed forward. After the data distribution, `t_mode()` is invoked and, as in event detectors, no v-section search is performed.

5.1.5 Tree-structured Decomposition

The tree-structure decomposition algorithm in [1] is used to scale input data in time. The resulting *coarser* data is passed to slaves working on the derived time scale. In the MLM, tree-structured decomposition is used to downsample data by a factor of 4. Downsampling by 4 gives an output block of 128 samples. Thus, multiscale event detection is performed on two time scales: the original *fine* time scale and the *coarse* scale (input downsampled by 4).

The input data is first convolved with a low pass filter [19], [20]. The convolution filter is in file *conv.tmp* and is loaded as a template at 0x3600 into the decomposer slave processors. The filtered data is then decimated in time by the `decimate()` subroutine. The problem with decimating by simply picking out every other sample is that it may cause regions of high-variation (v-sections) in the input data to not be fully represented in the downsampled data. See the discussion in [1] on how *adaptively selecting between resolving paths*, ensures that arbitrary shift in input data does not cause the decimator to underrepresent a v-section. The `decimate()` routine uses this technique to perform decimation in time, so that v-sections are well-represented in the downsampled data.

The tree-structured decomposition code transfers the time-scaled data to event detectors slaves searching on that time scale. The routine that performs this task must emulate the master board, since the code in the listening slaves remains unchanged. The subroutine is included here:

```

void master()
{
.
.
ioport2 = ioport2 | 0x40; /* p2-6 = 1: ioport1 input buff. disable */
ioport1 = 0;
t_ios0 = ios0;
wsr = 15;
ios0 = t_ios0 | 0x02; /* INT = HSO-1 */
wsr = 0;
WAIT_COMM
WAIT_COMM
t_ios0 = ios0;
wsr = 15;
ios0 = t_ios0 & 0xfd; /* INT = HSO-1 */
wsr = 0;

for(i = 0; i < DOUT_SIZE; i++)
{
WAIT_COMM
ioport1 = (unsigned char) (*(data_out+i) & 0x3F); /* 6 LSB */
t_ios0 = ios0;
wsr = 15;
ios0 = t_ios0 | 0x01; /* DAV = HSO-0 */
wsr = 0;
WAIT_COMM
t_ios0 = ios0;
wsr = 15;
ios0 = t_ios0 & 0xFE; /* DAV = HSO-0 */
wsr = 0;
WAIT_COMM
WAIT_COMM

```



```

ioport1 = (unsigned char) ((*data_out+i) & 0x0fc0) >> 6); /* 6 MSB */
t_ios0 = ios0;
wsr = 15;
ios0 = t_ios0 | 0x01; /* DAV = HSO-0 */
wsr = 0;
WAIT_COMM
t_ios0 = ios0;
wsr = 15;
ios0 = t_ios0 & 0xFE; /* DAV = HSO-0 */
wsr = 0;
WAIT_COMM
}
ioport1 = 0xff;
ioport2 = ioport2 & 0xbf; /* p2-6 = 0: ioport1 input buff. enable */
master_out++;
}

```

Observe that HSO-0 is used to provide the *DAV* signal and HSO-1 provides *INT*. Hence, if the *decomposer* slave is also used as an event detector as part of a load chain, only HSI/O lines 2 and 3 can participate in the chain. Note also that the structure is compatible with the NMI servicing routine of the receiving slaves. These slaves need no modification to work with the *decomposer* slaves rather than the master board.

5.1.6 Result Collation

Collaters also function around a structure called **load**, defined in software. As the collaters do not perform event detection, this structure is used primarily to hold the record sent by the event detectors and to be sent to the host PC upon request:

```

struct load{
unsigned char new_hit;      /* 1 means yes, 0 means no */
unsigned char used;        /* load is used? */
unsigned char got_proc;    /* processor which got the final vsecs */
unsigned char mach_no;     /* load ID */

```

```

unsigned char time_scale; /* time scale of identification */
unsigned char vs_loc_hi; /*offset from begin. of input block of last vs*/
unsigned char vs_loc_lo; /*offset from begin. of input block of last vs*/
unsigned char acq_time_hi; /* time of event */
unsigned char acq_time_lo; /* time of event */
}

```

There are two type of collaters in the MLM: the *normal collaters* and the *wide collaters*. The former accept results from four load chains on the four HSI lines. The latter can communicate with a total of eight chains by using Port 1 pins as additional I/O lines. Note that for the wide collater, Port 1 may not be used to talk to the master board. This is perfectly legal as the collaters do not process the input data anyway. The NMI interrupts still occur but no data is presented at Port 1 and the servicing routine is suitably abbreviated:

```

void nmi_master_int(void)
{
no_of_acq++;
}

```

For the wide collater, the four Port 1 pin pairs P1.0/P1.1, P1.2/P1.3, P1.4/P1.5, P1.6/P1.7 are used in a manner analagous to the HSI/O line pairs, to communicate with the event detector processors. Loads are assigned in the following order, to the HSI/O pairs and the Port 1 pin pairs:

```

Load L1 => HSI0 0
Load L2 => HSI0 1
Load L3 => HSI0 2
Load L4 => HSI0 3
Load L5 => P1 0/1
Load L6 => P1 2/3
Load L7 => P1 4/5
Load L8 => P1 6/7

```

It is important that the actual hardware hookup of load chains to the collater HSI and Port 1 lines is consistent with the loads attributed to each collater during collater initialization

in the Host PC Interface Program (see Section 5.2.9).

The collaters do not service interrupts on the HSI lines to get detection results. Instead they poll all the used HSI lines. If the HSI bit is set, the event detector processor is signalled to start communication of the identification record. This is received on the serial port and the data is placed in the structure of the load corresponding to the HSI/Port 1 line. The code for the wide collaters is used for normal collaters as well, by simply initializing loads 5–8 as not used. See the code in `p5_c1p.c` in Appendix D.

5.1.7 PC Communication

The host PC uploads the results of transient event detection from the collaters. It polls the collaters to check if a load has been identified on any time scale. The global variable "any_hit" in the collater code is set if a load has been identified on that time scale. The structures containing the identification record, etc., for each load are placed in memory as follows:

```
#pragma locate (11 = 0x3000)
#pragma locate (12 = 0x3010)
#pragma locate (13 = 0x3020)
#pragma locate (14 = 0x3030)
#pragma locate (15 = 0x3040)
#pragma locate (16 = 0x3050)
#pragma locate (17 = 0x3060)
#pragma locate (18 = 0x3070)
```

If the PC sees any_hit set to 1, it serially retrieves memory locations 0x3000, 0x3010, 0x3020,..., 0x3070 (`load.new_hit` elements of the load structures) to see which of the eight loads have been identified. It then proceeds to recover the entire record for each of those loads. Details of Collater-PC communication from the Host Interface perspective are given in Section 5.2.9.

In addition to collater communication, the Host PC also communicates with the other slaves to perform several important tasks as explained in Section 5.2. One feature of Slave-PC Communication implemented in the slave processor code is discussed here first. The event detector and decomposer slaves all have special global variables located in memory

locations 0x3500 through 0x3520. Some of these (e.g. `no_of_acq`, `new_acq` etc.) are important to the operations of the slave module. The rest were used for debugging/testing purposes. They may now be used to track the code execution of the slave. For example the pair of variables `sr_in/sr_out` are incremented, respectively, each time the slave module enters and leaves the subroutine performing serial communication with the collater. For example, if they are both seen to have a value of 8, it may be deduced that the slave has successfully performed eight serial transmissions of identification records. Suppose somehow the ninth transmission could not be completed successfully. Now `sr_in` will have the value 9 but `sr_out` would still be at 8. This information precisely identifies the nature of the problem to the user. Other debugging variables behave similarly. An annotated listing is included below.

```
#pragma locate (new_acq = 0x3500) /* = 1 if new data block just received */
#pragma locate (no_of_acq = 0x3502)/*no of input acquisitions made so far*/
#pragma locate (mode = 0x3504) /* = 1 if in template acquisition mode */
#pragma locate (temp_acq = 0x3506) /* acquisition no. frozen in temp mode*/
#pragma locate (temp_add = 0x3508)/*mem. add where change-of-mean occured*/
#pragma locate (hsi_avail_in = 0x3510) /* hsi_interrupt routine entered */
#pragma locate (hsi_avail_out = 0x3511) /* hsi_interrupt routine left */
#pragma locate (gotcha_in = 0x3512) /* next proc. comm. routine entered */
#pragma locate (gotcha_out = 0x3513) /* next proc. comm. routine left */
#pragma locate (euc_in = 0x3514) /* Euclidean Filtering routine entered */
#pragma locate (euc_out = 0x3515) /* Euclidean Filtering routine left */
#pragma locate (sr_in = 0x3516) /*Serial Comm to Collater routine entered*/
#pragma locate (sr_out = 0x3518) /*Serial Comm to Collater routine left*/
```

5.2 Software Design of the PC interface

The Host Interface for the MLM is implemented on a Pentium-based PC. Serial port COM1 is used to communicate with the slave modules. A PC I/O card was built and installed in the PC (see Appendix F). This is used to send the 8-bit processor ID to each slave board. It is also used to communicate with the master board to retrieve the most recently acquired data. Four handshaking bits and 16 data bits make up this communication channel.

A summary of the functions performed by the Host PC Code is given below:

- **Master board communication:** Retrieve the last eight acquisition blocks (about 20 sec. of acquired data) sent to the slaves. Data from all eight input channels is transferred to the PC.
- **Master board mode selection:** Place the master board in *sleep mode* when no data blocks are sent to the slaves or in *acquisition (or transfer) mode* in which data is sent to the slaves.
- **Processor selection:** Select a slave module for communication.
- **Load program code:** Download program code into the RAM of the selected processor.
- **Load templates:** Place in processor memory the templates to be searched by the selected slave.
- **Read memory locations:** Read specified memory locations, including Special Function Registers (SFRs). This feature is helpful in testing/debugging, as well as in other interface functions such as template acquisition.
- **Reset processors:** Reset a selected slave processor, or all MLM slaves.
- **Template acquisition:** Allow data for transient templates to be acquired and final templates to be constructed.
- **Poll collaters for results:** Allow user to identify certain slaves as collaters and poll these slaves for load identification results.
- **Display results graphically:** Display all hits graphically. Also display graphically the window of data acquired from the master board.
- **Update history.txt:** Maintain and update a file (history.txt) which contains a record of all load activity since the time the MLM PC Interface was invoked.

The software program that forms the Interface between the Host PC and the MLM slaves is actually implemented in two parts. One part resides on the Host PC and is called the Embedded Controller Monitor (ECM). (Intel's ECM for the 80C196 Evaluation Board [14]

may be regarded as a distant ancestor.) The other part is resident in Read Only Memory (ROM) of each MLM slave, and is known as the Reduced Instruction Set Monitor (RISM). The partitioning of the interface has several advantages. By placing a major portion of the Monitor code on the host PC, we ensure that the feature set of the user interface is not limited by the resources of the MLM slaves. By placing part of the program in the slave modules, we allow concurrent operation of the ECM and the MLM slaves: The user can trace and modify the state of a slave processor while it is running. The structure of each part is discussed separately. ECM is, of course, the more important part to understand, from a user's point-of-view.

5.2.1 RISM Structure

The RISM code used in the MLM interface is a modified version of the RISM code provided by Intel with the 80C196KC Evaluation Board [14]. The RISM is made up of about 300 bytes of 80C196 assembly code. It consists of a section of initialization where the slave module's external UART is configured and Interrupt initialization is performed. In addition there is an Interrupt Servicing Routine (ISR) that processes interrupt requests from the ECM program. The interface works as follows. The ECM sends a character to the selected slave. The character may be a command which the RISM ISR must recognize and carryout, or it may be a data byte to be placed in the slave's RAM. When the slave module UART receives the character, it interrupts the processor. This causes the ISR to be executed. The ISR reads in the character sent by ECM from the UART. It first determines if it represents a data or a command. If the character is greater than 0x1F or if the DLE (Data Load Enable) Flag is set, the character is treated as a data byte. If the character is a command, the ISR executes a case-jump to the section of code responsible for handling the command. The slave module then proceeds with normal code execution until it is interrupted again by the Host PC.

Here is a listing of the commands, and the characters that represent them, used by ECM and executed by RISM to control the working of the slave modules. The command names are self-explanatory.

SET_DLE_FLAG	0x00
TRANSMIT	0x02
READ_BYTE	0x04

Address (HEX)	After RESET	After REMAP
0000–00FF (as data)	Internal Registers	Internal Registers
0000–00FF (as code)	RISM EPROM	RISM EPROM
0100–1C00	Unused	Unused
1C00–1C10	RISM EPROM	RISM EPROM
1C10–1CFF	Unused	Unused
1D00–1DFF	RISM EPROM	RISM EPROM
1E00–1EFF	Ext. UART	Ext. UART
1F00–1FFF	Unused	Unused
2000–27FF	RISM EPROM	USER CODE/DATA RAM
2800–5FFF	USER CODE/DATA RAM	USER CODE/DATA RAM

Table 5.1: Memory Map of the Slave Module 80C196KC

READ_WORD	0x05
WRITE_BYTE	0x07
WRITE_WORD	0x08
LOAD_ADDRESS	0x0A
READ_PSW	0x0C
WRITE_PSW	0x0D
READ_SP	0x0E
WRITE_SP	0x0F
READ_PC	0x10
WRITE_PC	0x11
GO	0x12
HALT	0x13
REPORT_STAT	0x14
RESET	0x15

The command execution routines are short, so that the flow through the entire ISR is about 20 instructions. The serial communication, which occurs at 9600 baud, and the servicing of the command, thus take up very little time so that no real-time is lost to the slave processor, unless the user makes ECM actively interrupt the slaves.

When the PC interrupts the 80C196, the processor is, in general, executing program code from the RAM. It must switch to executing ISR code in the ROM, possibly at the same memory address as the RAM code. How is this memory map implemented and managed?

The Decode PAL (referred to as `BUS_CON` in Appendices B and C) outputs the Chip Select signals to the ROM, RAM and UART, depending on the memory address to be accessed by the microcontroller. However certain memory ranges are common to both the ROM and the RAM. For instance, address range 0x2000 to 0x2800 in the ROM or the RAM may both contain executable code. In a certain mode, code must be fetched from the ROM, in another from the RAM. The PAL outputs a bit called the MAP bit. If the MAP bit is not set, the memory ranges common to both the ROM and RAM are fetched from the ROM. The RISM is said to be in “`not_user mode`”. If MAP is set, and instructions are accessed from the overlapping memory ranges, code is fetched from the RAM, and the RISM is in “`user mode`”. On power up, the MAP bit is reset and code is executed from ROM (which makes sense since there *is no code in the RAM yet.*) Once code has been downloaded into RAM and the `user mode` has been entered, the RAM code is fetched and executed. Even in user mode some memory locations are mapped exclusively to the ROM e.g. 0x00 to 0xFF in the ROM contains the ISR. When an interrupt from the PC is received, memory access is initiated from these locations and the ROM is selected by the PAL. A summary of the memory map is given in Table 5.1.

Some important changes were made to the Intel RISM structure to obtain the RISM used for the MLM. The interrupt dedicated to MLM-PC communication was the P2-2 interrupt rather than the NMI. Thus the interrupt vector initialization is different for the MLM RISM where the P2-2 interrupt vectors to the ISR and the NMI is reserved for master-slave communication. The PSW initialization code is also modified, as the lower byte of the PSW is the `INT_MASK` register which is set to a different value for the MLM than the 80C196 Evaluation Board described in [14]. In order to accommodate these changes some assembly code had to be rewritten and a little code added to the existing RISM of the Evaluation Board. The changes made to the RISM assembly code are discussed in Appendix E. A detailed discussion of the original RISM can be found in [14], the 80C196EVB Manual. The entire RISM code is also included in this handbook.

5.2.2 ECM Program Structure

The code resident in the PC was developed using a Microsoft C compiler, version 6.0 [21]. The executable file is called ECM (for Embedded Controller Monitor) and was designed and implemented completely and exclusively for this thesis. The program is a DOS-based menu-


```
*****          MAIN MENU          *****  
  
1. Slave Communication  
2. Slave Diagnostic/Echo  
3. Slave Processor Selection  
4. Data from Master  
5. Master Acquisition Mode Setup  
6. Slave Mode Selection  
7. Collater Communication  
8. Display Channel Data  
9. Initialize Slave Processors  
E. Exit ECM
```

Figure 5.1: ECM Main Menu

driven interface. The code is listed in Appendix E and should be referenced throughout the rest of the chapter when the menu selections and the functions they perform are discussed.

The top-level menu selections for ECM are shown in Figure 5.1. Several menu selections lead to submenus. Main menu choice 1, for instance opens up a submenu for functions concerned with slave communications (e.g., downloading code). Certain choices in slave communication present further menu selections. This heirarchical structure is logically constructed and easy to follow. In general, the ECM code is robust and user-friendly: There is always the option to exit from submenus, and before any major operation is performed the user is given the option to recall their selection. Main menu choices 4, 5 and 8 are largely concerned with controlling and communicating with the master board. Selection 2 is used for debugging purposes only and is therefore not important for the MLM user. Selections 1, 3, 6, 7, and 9 control the slave modules.

We first consider master board communication and control.

5.2.3 Communicating with the Master Board

An important function of the PC Interface is retrieving the last eight blocks of data transferred to the slave processors by the master board, when an event is detected or input data

is requested by the user. As explained in Chapter 3, the master board not only sends blocks of digitized data to the slaves, it also stores upto eight blocks of data in a circular buffer implemented using RAMs, counters and PALs. The PC interface accesses this data via a PC I/O card (see Appendix G). Details of the handshaking protocol are given in Chapter 3.

To invoke this function, select 4 (*Data From Master*) in the main menu and initiate data acquisition. ECM uploads the data stored in the master board PC interface RAMs, and places each channel's data separately in files named **ch1.dat**, **ch2.dat**,..., **ch7.dat**, **ch8.dat**. Each of these files has 4096 (= 8 x 512) samples of input data, spanning about 20 seconds of acquisition time — the input sampling rate is set at 200Hz.

The data on channels 1, 2, 3, and 4 may be displayed using the main menu choice 8 (*Display Channel Data*) . When selection 8 is invoked, ECM graphs the data of four channels contained in the respective “.dat” file, as shown in Figure 5.2. It automatically scales the y-axis depending on the values of the data points in the files. Channel 1 data is in the upper right plot, channel 2 is plotted on the upper left corner. In the lower half, from left to right, are channels 3 and 4. The *MLM Console* may be ignored while viewing channel data from the master board.

At present this large window of data, containing the last 20 seconds of sampled input, available from the master board is used primarily for template acquisition, as discussed later in the chapter. This data is also used by ECM whenever a load is identified by the MLM. The ECM informs the user of the loads identified (lists them in the *MLM Console*), retrieves the recently acquired data from the master board, and displays channels 1 through 4 in the same manner as menu selection 8. The ECM essentially invokes selections 4 (to retrieve data) and 8 (to display data) of the main menu.

Other applications of this data bank may be developed as future work is done on the MLM platform. It would, for instance, be necessary to look at the acquired data for the purposes of diagnostic load monitoring. Similarly, if the objective was to look at higher harmonics of current to track down *power quality offenders*, this feature would be very useful.

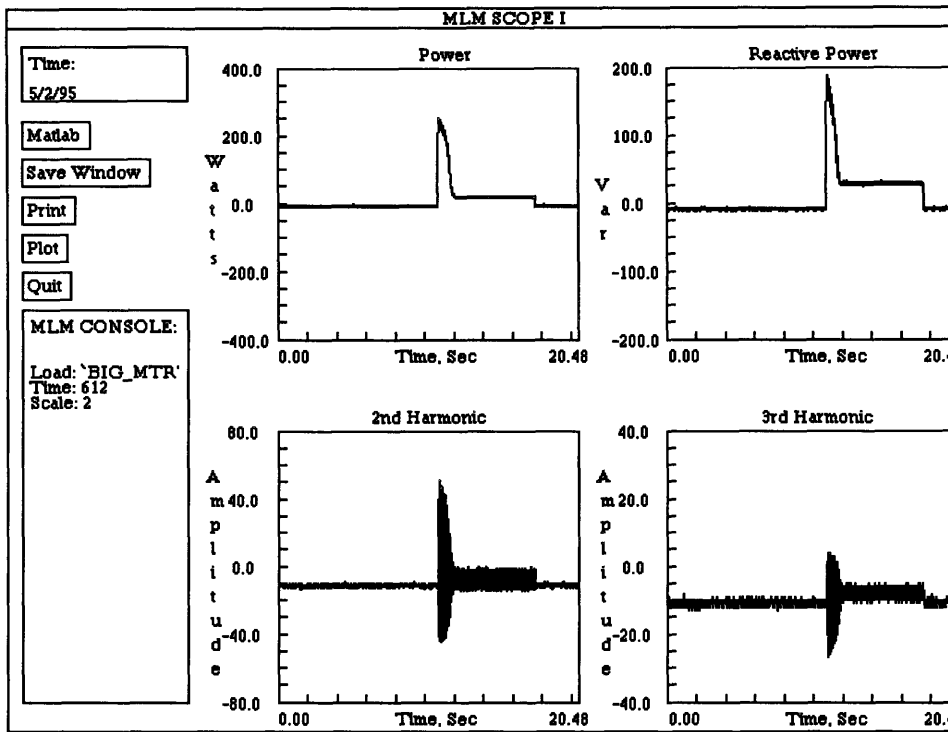


Figure 5.2: Displaying Channel Data

5.2.4 Master Board Mode Selection

While the ECM controls the MLM largely by controlling the slave processors, there is one important way in which it influences the functioning of the MLM master board. If option 5 (*Master Acquisition Mode Setup*) in the main menu is selected, the submenu shown in Figure 5.3 is brought up. The current state of the master board (“sleep mode” or “acquisition mode”) is displayed above the menu selections. The user may then use the choices to set the mode and exit back to the main menu.

In *sleep mode* the master board continues to acquire data from the analog preprocessor, stores them in the data banks as well as the PC interface memory. However, it does not ship any data to the slaves. A single control line goes from the Host PC (via the PC I/O card) to the “Transfer PALs” and inhibits the Interrupt signal from being sent to the slave processors. In this mode therefore, the MLM as a whole is essentially asleep. The master board may however, be accessed, to provide 20 seconds of data at any time: Sampling of the input streams never stops.

In the *acquisition mode*, the PC allows the master board to send data to the slave mod-

```
***** MASTER MODE SET UP *****

** Master is in Sleep Mode **

Select Function:

1. Place Master Board in Acquisition Mode
2. Place Master Board in Sleep Mode
3. Exit
```

Figure 5.3: Master Acquisition Mode Submenu

ules. The slaves can then process the data and report load identifications back to the host PC. It is imperative to “activate” the system *only after all slaves have been initialized* e.g., the program code has been downloaded. Otherwise the slave modules receive Non Maskable Interrupts from the master board without having the interrupt servicing routine available to them in their program memory. This may place them in an undefined/undesired state, necessitating a “hard” reset. It is also important *not to communicate with the slaves* when the system is active. For instance, if program code were to be downloaded into a slave while the master board was in *acquisition mode*, the serial communication of program code could be interrupted by an NMI interrupt from the master. This could potentially misplace or fragment the program code in the slave processor so that when the code is run, the processor may enter an undesirable state.

Having dealt with the the ECM master board interface, we turn to the menu options that implement the slave interface, beginning with the process of slave selection.

5.2.5 Selecting a Slave Processor for Communication

To select a slave, choose option 3 (**Slave Processor Selection**) in the main menu. ECM prompts the user for the processor ID which it expects to be a number in the range 0 to 255, inclusive. It also allows the user to re-enter the ID if a mistake is made. Once a processor is selected, it will be the processor addressed in all menu and submenu selections dealing with slave communication. It is important to select a slave processor when the ECM program is first started, so as to be sure which processor is being addressed.

The mechanism to carry out this selection is simple. Once the user has entered a valid processor ID, the ID byte is transmitted to a memory-mapped register (LS374) on the PC I/O card (see Appendix F). The outputs of this register go to the “glue logic” on each slave board, where it is compared to the hardwired board ID. Based on these comparisons, one slave module is connected directly to the PC serial port.

Once a slave has been addressed, we can select item 1 (**Slave Communication**) in the main menu and download code, templates, etc. Before we address this menu selection however, we must explain the procedures performed to get a final program code file and to extract templates from raw data. These topics precede our discussion of ECM's slave interface.

5.2.6 Program Code Output Format

The 80C196KC compiler, IC-96, allows C program code to be compiled and outputted in a format used by the 80C196 evaluation board's Host PC Interface to download programs to the 80C196 on the evaluation board (see Appendix D for a listing of the batch file, *cc.bat*, that invokes *ic-96*). This format is, however, of little use for us as we are using a host interface developed exclusively for the MLM. The “output-to-hex converter” utility [13] is used to achieve a format that the ECM can recognize and work with. This converter is invoked by the “OH” DOS line command:

```
C:\> OH myfile.OUT TO myfile.HEX
```

The .HEX file contains the program code as well as its RAM address etc., in ascii format so that ECM can use these files to place code in the slaves. Part of a sample .HEX file is listed below:

:03208000E7DB0398
:07245E00A1640018EF38FC37
:02203E008320FD
:10208300F4C81CC81EA30100351C4501001C1EC357
:102093000100351ECC1ECC1CF5F069020018C81ACD
:1020A300A0181AC824111CC70116351CC7011735FF
:1020B3001C2A2522189B01013000D702203B510224

5.2.7 Template Collection

One of the more important tasks of the Host PC is the collection of templates of v-sections that may be sent to slave modules for use in pattern search. There are two ways of producing raw template data files using ECM.

If the templates are on the original time scale, data is retrieved from the *master board* and placed in files *ch1.dat*,..., *ch8.dat*. (This method can, in theory, also be applied to extract templates on other time scales. However, it is easier to extract templates on derived time scales directly from the slaves performing event detection). The method is simple: Once the load of interest is turned on, main menu selection 4 is used to acquire the recently sampled data. The transients occurring on the various input streams due to the load's activity are thus captured and placed in the respective ".dat" files. The user quits ECM and starts a mathematics/graphics package (in our case, MATLAB). The ".dat" files are loaded into MATLAB (or equivalent). Each stream is considered in turn and the v-sections are first identified and then isolated into separate text files. To keep things clear a simple protocol was developed for naming these template files. Suppose the load in question was a Motor. And suppose we were to isolate two v-sections in stream 1 (P) and two in stream 2 (Q). The names for the files containing the raw data points for these four templates are given in Table 5.2. The first letter identifies the load, the second letter identifies the data stream, the number denotes the sequence in which the v-sections occur in the input stream (1st, 2nd etc.), and the ".RAW" extension signifies that this is a file containing raw template data.

The second method of capturing and partitioning raw data into v-section files accesses the slave modules directly. In this procedure, the slave modes are placed in *template acquisition mode*, using main menu selection 6. The submenu for determining the mode of

Content	File Name
1st Motor v-section in P	MP1.RAW
2nd Motor v-section in P	MP2.RAW
1st Motor v-section in Q	MQ1.RAW
2nd Motor v-section in Q	MQ2.RAW

Table 5.2: Naming Scheme for Raw V-section Template Files

***** SLAVE MODE SET UP *****

Select Function:

1. Place Slave Modules in Template Acquisition Mode
2. Place Slave Modules in Normal Function Mode
3. Reset All Slave Processors
4. Exit

Figure 5.4: Slave Mode Selection Submenu

operation of slave processors is given in Figure 5.4. Selecting 1 in this menu places all processors in template acquisition mode. The processors keep acquiring data from the master (the master must be in *acquisition mode*) but do not process it; i.e., no event detection takes place (tree-structured decomposition does go on so that the slaves on different time scales do get scaled data from the decomposer slaves). The slaves simply wait for a change of mean in their input stream. When this change occurs, the slaves freeze all operations and acquire no further data. The window of 512 bytes in which the mean changed is thus preserved in each slaves memory. As this change occurred due to the activity of the load whose templates are being collected, the transient causing the change would be captured in this last acquired block of data. Note that all slaves performing event detection or tree-structured decomposition follow this behaviour in template acquisition mode. This means

that event detector slaves on derived time scales freeze the transients they will be searching for, just like event detectors accepting input data directly from the master board.

Once the load has been turned on and its transient v-sections captured, the slave communication menu is entered (see Figure 5.5) and the relevant range of memory is read (details in the following subsection) and dumped to a file. This file is then studied by the user and v-sections are extracted from the data to form the raw v-section files.

The final step in template collection is conditioning the “.RAW” to produce the final “.TMP” file to be loaded into slave memory. The main step is subtracting the mean (dc value) of the raw template from each point, i.e. ac-coupling the template, as this is what the euclidean filter does to the input stream it searches. ECM provides this conversion capability as a submenu option.

Figure 5.5 shows the Slave Communication submenu that is accessed by choosing option 1 of the main menu. To convert a .RAW file to a .TMP file, choose option 2 (**Template Management**). The menu shown in Figure 5.6 is displayed. Choose option 2 (**Convert Raw Data to Template**) within this menu. The system will prompt the user for the .RAW file name, as well as the final .TMP file name. The names, along with the extensions, must be entered. If the .RAW file exists and has the data in the correct format, the ac-coupling is performed and a .TMP file is generated. Otherwise a message is printed explaining what went wrong.

As an illustration of the correct format, consider the file MP1.RAW given below:

```
:FFF5
:FFF2
:FFF2
:002C
:004C
:0079
:0084
:0086
END
```

```
/* This is the 1st raw v-section for Motor in P. The data below was
```


***** Normal Mode of Execution *****

1. Load File
2. Template Management
3. Go
4. Halt
5. Read Range
6. Poke Memory
7. Reset Processor 0
8. Reset Processor 0 and Remap to User
9. Quit

Figure 5.5: Slave Communication Submenu

uploaded from a slave and edited for v-sections. */

```
ADDR:                16 bytes of Data (LSB First)
-----
5AF0:   F4 FF F5 FF F4 FF F4 FF F4 FF F5 FF F4 FF F4 FF
5B00:   F6 FF F2 FF F2 FF 2C 00 4C 00 79 00 84 00 86 00
5B10:   94 00 8B 00 95 00 8B 00 8D 00 95 00 88 00 91 00
5B20:   88 00 8B 00 8F 00 82 00 89 00 84 00 8A 00 88 00
-----
```

The data values in MP1.RAW are in hexadecimal, with each point preceded by a colon (:). The end of the v-section is denoted by the string “END” as shown. Below that remarks, original data points etc. are placed (as shown) for future reference. Regardless of which method is taken to generate the raw v-sections, this format is expected by ECM. If MATLAB is used in method 1 above, to generate the v-section files, the output is given in exponential form. This can be translated into the desired format by using the executable “TRANS2” available on the host PC. If the v-sections are hand-picked using the slaves in template collection mode, it is usually easier to manually edit the v-section file to get them

***** Template Management *****

Select Function:

1. Load Template File
2. Convert Raw Data to Template
3. Exit

Figure 5.6: Template Management Submenu

in the correct format.

The MP1.TMP file generated by ECM (from MP1.RAW) is as follows:

```
:FFBB
:FFB8
:FFB8
:FFF2
:0012
:003F
:004A
:004C
END
```

Statistics on Template:

Size = 8

Sum = 468

DC value = 58

ECM generates statistics such as template size, sum of data points, and mean value for reference. This file may now be downloaded into a slaves RAM using the ECM slave communication options.

5.2.8 Communicating with the Slave Processors

Having explained the process of getting final output files for program code and templates, we now discuss the ECM menu selections that perform the slave communication.

Figure 5.5 shows the submenu within main menu option 1 (**Slave Communication**). The most important function (and the most commonly used!) is item 1 (**Load File**). When this is selected, ECM prompts for a file name. As explained earlier this is expected to be in Intel HEX format. The program code is then serially downloaded into slave memory.

Option 2 (**Template Management**) displays the submenu shown in Figure 5.6. This submenu allows users to perform two important operations: Loading templates into slave memory using selection 1, and converting raw templates to the final ac-coupled form that is used by the slaves. When option 1 is selected, ECM asks for the .TMP file to be loaded into slave memory. It also requires the address at which the template must be placed. Option 2 and its usage have been discussed in the previous section.

Once program code and template data has been loaded into its RAM, a slave is ready to start program execution. The 3rd option in the Slave Communication menu (in Figure 5.5), **GO**, starts code execution in the slave. Option 4 (**HALT**) may be used to stop execution. If the slave is commanded to run again, code execution will be restarted from the instruction at which the processor was halted.

Option 5 (**READ RANGE**) in the slave communication submenu allows the user to retrieve a specified range of RAM locations, display the range, and dump the data to a file. When option 5 is selected, the system prompts for the start and end addresses of the memory range to be read. If the address range size is valid (≥ 0 and ≤ 1024), the specified locations are retrieved from processor memory and displayed. The user is next asked if the data is to be dumped to a file. If the option is availed, a file with the specified name is opened and data is placed in it. The data is placed in the file in the same manner as it is displayed in the ECM environment:

ADDR: 16 bytes of Data (LSB First)

```
5A90:  F5 FF F4 FF F5 FF F6 FF F6 FF F4 FF F6 FF F4 FF
5AA0:  F4 FF F6 FF F6 FF F2 FF F4 FF F7 FF F5 FF F4 FF
5AB0:  F6 FF F4 FF F5 FF F6 FF F5 FF F4 FF F5 FF F6 FF
5AC0:  F4 FF F4 FF F6 FF F4 FF F0 FF F5 FF F5 FF F2 FF
5AD0:  F5 FF F4 FF F6 FF F5 FF F8 FF F3 FF F3 FF F5 FF
5AE0:  F6 FF F3 FF F3 FF F5 FF F6 FF F4 FF F5 FF F5 FF
5AF0:  F4 FF F5 FF F4 FF F4 FF F4 FF F5 FF F4 FF F4 FF
5B00:  F6 FF F2 FF F2 FF 2C 00 4C 00 79 00 84 00 86 00
5B10:  94 00 8B 00 95 00 8B 00 8D 00 95 00 88 00 91 00
```

The data is displayed byte-wise. If the values are words, then the LSB is in the even memory location, and is displayed *before* the MSB in the odd memory location. A header is included for convenience in both the ECM display and the output file.

The read option is most useful in acquiring templates from the slave processors. It may also be used to verify that templates have been correctly placed in memory. Another use is to check the important memory locations 0x3500 to 0x3520 (as described in Section 5.1.7) to see if the slave is behaving properly.

Option 6 gives another easy way to check if the slave processor is in a valid state. The menu it brings up is in Figure 5.7. It allows the user to read the value of the Stack Pointer (SP), the Program Counter (PC) and the Processor Status Word (PSW). In the Reset state, the PC equals 0x2080, and the SP is at 0x100. Under normal program execution, valid ranges for the PC are:

For Event Detectors : $0x2085 \leq PC \leq 0x2600$.

For T.S.-Decomposers: $0x2085 \leq PC \leq 0x2600$.

For Collaters: $0x2085 \leq PC \leq 0x2300$.

The SP may have a value between 0x100 and 0x140 during normal code execution. The lower byte of the PSW is the content of the INT_MASK register, the upper byte contains various flags (e.g. the “overflow” flag). The expected value of INT_MASK can be compared

******* Read Special Function Registers *******

- 1. Fetch SP**
- 2. Fetch PC**
- 3. Fetch PSW**
- 4. Exit**

Figure 5.7: Special Function Register Submenu

to the lower byte of PSW to determine if the processor is in normal execution state.

Options 7 and 8 issue a software reset command to the slave processor. In general, option 8 should be used to reset processors, as the processor is also user-mapped i.e. is ready to accept program code from the PC or execute code already in memory (see 5.2.1).

These are the menu options within the Slave Communication Menu. Most of the Communication between the PC and the slaves is carried out through them. Other ECM main menu options give additional features to control slave modules. Main menu option 2 was used during the design of the software interface as a diagnostic tool, and is of no major use at present. Option 3, which selects a slave processor for communication, has been discussed earlier. Option 6 leads to the submenu shown in Figure 5.4. Here the mode of operation of all slave processors may be set using options 1 and 2, as discussed in Section 5.2.7. Option 3 in this submenu allows all processors to be simultaneously reset and is preferable to a *hard reset* of the system. Because it toggles the *RESET* input pin of each slave 80C196 processor directly, as opposed to issuing a software RESET instruction, it is a more powerful option than software reset.

5.2.9 Collater Communication

We have discussed main menu options 1 through 6, and option 8. These options allow us to interface with the master board as well as configure the slave processors to perform transient event detection. Main menu item 7 allows us to upload the results of transient search from the collater processors. When Option 7 is selected, ECM asks the user if collater initialization needs to be performed. Since ECM does not automatically know which of the slaves is functioning as a collater and what loads it is looking at, this initialization must be performed whenever ECM is invoked. The user does not need to key in the collater IDs or the loads whose identification is reported to each collater. ECM looks at a file "loadname.txt" to determine all that by itself. The file "loadname.txt" for the 16 processor MLM prototype (see Chapter 6) developed for this thesis is listed below:

```
:end
:end
:end
:end
:end
:end
RAPID
MOTOR
COMP
INSTANT
LIGHT
:end
:end
BIG_MTR
:end
:end
:end
:end
:end
:end
:end
```

```
:end
:end
```

There are 16 “:end” statements in the file corresponding to the 16 slave processors. If a slave is a collater, the names of its loads are entered before its “:end” statement. Thus, in the file above, processors 5 and 7 (remember that the first processor has ID number 0), are designated collaters. Processor 5 is responsible for five loads, while 7 has one load, BIG_MTR assigned to it. To *de-activate* a collater using the loadname.txt file, simply remove all the load names associated with that processor in the file. For instance, if the BIG_MTR entry is removed from the file, ECM will not consider processor 7 a collater anymore and will not poll it for results.

ECM uses a double array called “packet[][]” to store the initialization information. The first dimension corresponds to the collater and the second dimension to the loads per collater. To continue with our sample “loadname.txt” file, processor 5 is designated the first collater and “INSTANT” is the 4th load assigned to it. Hence array entry packet[0][3] contains information on this load. Similarly packet[1][0] is the entry associated with load 0, “BIG MTR”, on collater 1 (processor 7). Each entry of the array is actually a structure that houses the associated load name as well as all identification information retrieved from the collater when load activity is reported. The structure is as follows:

```
struct load_hit{
unsigned char ld_hit; /* load identified */
char name[10];      /* name of load */
unsigned char load_id; /* load id */
unsigned char scale; /* time scale of identification */
unsigned char hi_loc; /* offset from begin. of input block for last vs*/
unsigned char lo_loc; /* offset from begin. of input block for last vs*/
unsigned char hi_time; /* time of event */
unsigned char lo_time; /* time of event */
unsigned int cur_time; /* current acquisition no (time) */
}
```

Note that the structure parallels the structure in the collater program code that holds the load identification packet.

Once collater initialization is performed, ECM may be placed in “Collater Communication Mode” in which the system waits for a load to be identified. It polls the designated collaters on a round-robin basis. The user can break out of this mode by pressing any key. If the PC sees that a load has been identified, it uploads the identification packet from the collater and places it in the correct entry of array “packet[][]”. It resets the “hit flag” of the collater (so that the same identification is not reported more than once) and reports the event to the user. Next, it retrieves the last 12 seconds of data from the master board and displays it graphically just as main menu option 8 does (see Figure 5.2). The MLM Console displays the load(s) identified, the time scale, and the identification “time” (whose units are the number of input data blocks sent to the slaves by the master board). ECM also updates a file “history.txt” with the information displayed in the MLM Console. To reset this recording of load detections, simply erase history.txt. ECM will open a new history.txt file and start afresh. A portion of a sample history.txt is listed here:

Load: 'INSTANT'

Time: 365

Scale: 1

Load: 'RAPID'

Time: 424

Scale: 1

Load: 'LIGHT'

Time: 457

Scale: 1

Load: 'RAPID'

Time: 1375

Scale: 1

Load: 'MOTOR'

Time: 1375

Scale: 1

5.2.10 Automatic Initialization of Slave Processors

Downloading code and templates into 16 processors and starting them up manually is not an enviable task. ECM gives the option of automatically performing all initializations in Selection 9 of the main menu. When this option is invoked, all processors are reset and each processor is selected for initialization in turn: program code is placed according to directions in file "procfile.txt"; templates are loaded according to the information in "proctemp.txt". Procfile.txt is a sequential listing of the ".HEX" files that are to be placed in processors 0,1,2,... in that order. A sample procfile.txt for initializing 16 processors is given below:

p0_e1p.hex
p1_e1q.hex
p2_e13a.hex
p3_e13.hex
p4_e1p.hex
p5_c1p.hex
p6_e1q.hex
p7_c2q.hex
p8_d1p.hex

p9_d1q.hex
p10_e2p.hex
p11_e2q.hex
p7_c2q.hex
p7_c2q.hex
p7_c2q.hex
p7_c2q.hex

Note the nomenclature of the files containing the program code. For instance, “p0_e1p” signifies that this file has program code for Processor 0 (p0), which performs event detection on time scale 1 (e1) on data stream P (p). Similarly “p7_c2q” states that the file contains code for processor 7 (p7), which is a collater on time scale 2 (c2), whose input stream is Q (q). And “p8_d1p” is the name for the file containing code for processor 8, performing tree-structured decomposition on scale 1, on stream P. There are occasional exceptions to the pattern such as “p2_e13a.hex”, where the extra “a” at the end denotes “asymmetrical code” i.e. this file does not contain the usual event detector code to be found in other event detector (“..e..”) files.

The “proctemp.txt” file contains a sequential listing of all templates in each processor. A portion of a proctemp.txt file is included here for illustration:

rp1.tmp
rp2.tmp
mp1.tmp
mp2.tmp
mp2.tmp
cp1.tmp
mp2.tmp
ip1.tmp
ip2.tmp
:end
rq1.tmp
mq1.tmp
mq2.tmp

```
:end  
r31.tmp  
mq2.tmp  
mq2.tmp  
i3.tmp  
:end  
c31.tmp  
c31.tmp  
:end
```

The “:end” statements separate the template file names for different processors. In the above sample, processor 0 is initialized with 9 templates, processor 1 has three templates, processor 2 has four templates and so on. For each processor, the templates are placed in memory in the order that they are listed in “proctemp.txt”, starting at location 0x3600 and stored at intervals of 0x200 (i.e. 0x3600, 0x3800, 0x3a00,...), as described in Section 5.1.

Chapter 6

Prototype Construction and Results

In this chapter we present the two prototype Multiprocessing Load Monitors constructed for this thesis. The development cycle is recapitulated, the configuration of each prototype is explained and a detailed discussion is made of the results collected.

6.1 Development Cycle for the MLM

Before describing the configuration of the two prototype Multiprocessing Load Monitor built as part of this thesis, a summary of their development is given. After developing the abstract MLM model, the first step was choosing a versatile yet inexpensive processor. Having decided on the 80C196KC microcontroller, a prototype of the slave module was made on a breadboard. At the same time the master board was also prototyped on a breadboard. Individual functionality and communication between the boards were tested and the designs were finalized. Logic level schematics were drawn in PADS to capture the design details. These were used to lay out the actual printed circuit boards (PCBs) and the layouts were sent for manufacturing. Software development was also begun at this point. The manufactured boards were populated with components and tested. The Host PC Interface's development was also undertaken. The code for the processors was tested out on the hardware. Once the software and hardware were verified to work correctly, the slave boards were configured to form the prototypes. The PC Interface was finalized and the prototypes were tested thoroughly on actual loads. The slave processor code, the

Module	Function	Time Scale	Input Stream	Input Source
M0	Pattern Search	1	P	Master Board
M1	Pattern Search	1	Q	Master Board
M2	Pattern Search	1	3P	Master Board
M3	Pattern Search	1	P	Master Board
M4	Decomposition	1	P	Master Board
M5	Collation	2	P	Master Board
M6	Collation	1	P	Master Board
M7	Pattern Search	2	P	Slave M4

Table 6.1: Slave Module Configuration in the MLM-8S

PAL code, and the ECM code were fine-tuned during this experimentation. Results of load identification with the multiscale transient event detection algorithm implemented on the MLM prototypes were collected. The results of this effort are described in the next four sections, and Appendix I presents a pictorial genesis of the MLM prototypes to go along with the discussion here.

6.2 Prototype I : MLM-8S

The two prototypes constructed differ only in the number of slave modules present. The smaller prototype, known as the MLM-8S, consists of a master board and two slave boards (8 slave processors). The three boards are stacked in a rack-mount chassis and powered by two power supplies. The assembly goes into a cabinet that also holds the Host PC.

Slave board 1 houses slave modules M0 through M3. These all perform transient detection on the original time scale on data streams P, Q and 3P. The second slave board houses M4 which performs tree-structured decomposition, M7 which is an event detector on the new time scale, and M5 and M6, the collaters for the two time scales. A summary of the slave modules' function and interconnection details are given in Tables 6.1 and 6.2. In addition to the HSI/O interconnections, each collater's RXD line is connected to the TXD lines of the last processors of the load chains listed below:

First Module	HSIO Line	Second Module	HSIO Line
M0	HSIO 0	M1	HSIO 0
M0	HSIO 1	M1	HSIO 1
M0	HSIO 2	M3	HSIO 0
M1	HSIO 2	M2	HSIO 2
M1	HSIO 3	M2	HSIO 3
M2	HSIO 0	M6	HSIO 0
M2	HSIO 1	M6	HSIO 1
M3	HSIO 2	M6	HSIO 2
M3	HSIO 3	M6	HSIO 3
M7	HSIO 0	M5	HSIO 0
M7	HSIO 1	M5	HSIO 1
M4	HSIO 3	M5	HSIO 3

Table 6.2: Slave-Slave Interconnections for MLM-8S

M2 (TXD) => M6 (RXD)

M3 (TXD) => M6 (RXD)

M7 (TXD) => M5 (RXD)

M4 (TXD) => M5 (RXD)

MLM-8S monitored three of the six loads available for testing. Details of the test stand and the loads monitored are given in Section 6.4. The load chains implemented in the MLM-8S are listed below, with the collater for each chain shown at the end, in square brackets:

MOTOR: M0(P)→ M1(Q)→ M2(3P) [→ M6]

INSTANT START LAMPS: M0(P)→ M1(Q)→ M2(3P) [→ M6]

INCANDESCENT LIGHT BULBS: M3(P) [→ M6]

The slave modules also make available to the user the following load chains, presently unused:

M3(P) [→ M6]

M7(P) [→ M5]

M7(P) [→ M5]

M7(P) [→ M5]

M4(P) [→ M5]

Module	Function	Time Scale	Input Stream	Input Source
M0	Pattern Search	1	P	Master Board
M1	Pattern Search	1	Q	Master Board
M2	Pattern Search	1	3P	Master Board
M3	Pattern Search	1	3P	Master Board
M4	Pattern Search	1	P	Master Board
M5	Collation	1	P	Master Board
M6	Pattern Search	1	P	Master Board
M7	Collation	2	3P	Master Board
M8	Decomposition	1	P	Master Board
M9	Decomposition	1	Q	Master Board
M10	Pattern Search	2	P	Slave M8
M11	Pattern Search	2	Q	Slave M9
M12	Pattern Search	1	P	Master Board
M13	Pattern Search	1	Q	Master Board
M14	Pattern Search	1	3P	Master Board
M15	Collation	1	3P	Master Board

Table 6.3: Slave Module Configuration in the MLM-16S

6.3 Prototype II : MLM-16S

The second prototype constructed, the larger of the two, is known as the MLM-16S. It consists of a master board and four slave boards giving a total of 16 slave processors. The five boards are stacked in two rack-mount chasses which, together with the power supplies, go into the cabinet that holds a host PC.

Slave board 1 houses slave modules M0 through M3. These all perform transient detection on the original time scale on P, Q and 3P. The second slave board houses M4 and M6, which are also event detectors, and M5 and M7, the *wide* collaters for the two time scales currently implemented in MLM-16S. Slave board 3 holds slave modules M8 and M9, performing tree-structured decomposition on P and Q respectively. M10 and M11 are the event detectors on the coarse scale which receive data, downsampled by 4, from M8 and M9. The fourth slave board consists of slave modules M12 through M15. Modules M12, M13, and M14 perform transient detection on the fine time scale on P, Q and 3P. M15 is a *normal* collater that takes event detection results from M12..14. A summary of the slave modules' function and interconnection details is given in Tables 6.3 and 6.4.

In addition to the HSIO interconnections each collater's RXD line is connected to the

First Module	HSIO Line	Second Module	HSIO/Port1 Line
M0	HSIO 0	M1	HSIO 0
M0	HSIO 1	M1	HSIO 1
M0	HSIO 2	M3	HSIO 2
M0	HSIO 3	M2	HSIO 3
M1	HSIO 2	M2	HSIO 2
M2	HSIO 0	M5	HSIO 0
M2	HSIO 1	M5	HSIO 3
M3	HSIO 0	M5	HSIO 2
M4	HSIO 0	M6	HSIO 0
M4	HSIO 1	M6	HSIO 1
M6	HSIO 2	M5	HSIO 1
M6	HSIO 3	M5	P1.0/1.1
M4	HSIO 2	M5	P1.2/1.3
M10	HSIO 0	M11	HSIO 0
M10	HSIO 1	M11	HSIO 1
M10	HSIO 2	M7	HSIO 2
M11	HSIO 2	M7	HSIO 0
M11	HSIO 3	M7	HSIO 1
M12	HSIO 0	M13	HSIO 0
M12	HSIO 1	M13	HSIO 1
M12	HSIO 2	M14	HSIO 0
M12	HSIO 3	M15	HSIO 0
M13	HSIO 2	M15	HSIO 1
M13	HSIO 3	M14	HSIO 1
M14	HSIO 2	M15	HSIO 2
M14	HSIO 3	M15	HSIO 3

Table 6.4: Slave-Slave Interconnections for MLM-16S

TXD lines of the last processors of the load chains listed below:

M2 (TXD) => M5 (RXD)

M3 (TXD) => M5 (RXD)

M6 (TXD) => M5 (RXD)

M4 (TXD) => M5 (RXD)

M9 (TXD) => M7 (RXD)

M11 (TXD) => M7 (RXD)

M14 (TXD) => M15 (RXD)

M13 (TXD) => M15 (RXD)

M12 (TXD) => M15 (RXD)

MLM-16S monitored all six loads available for testing. Details of the test stand and the loads monitored are given in Section 6.4. Load chains implemented in the MLM-16S are listed below, with the collater for each chain shown at the end, in square brackets:

INSTANT START LAMPS: M0(P)→ M2(3P) [→ M5]

RAPID START LAMPS: M0(P)→ M1(Q)→ M2(3P) [→ M5]

COMPUTER: M0(P)→ M3(3P) [→ M5]

SMALL MOTOR: M4(P)→ M6(Q) [→ M5]

INCANDESCENT LIGHT BULBS: M4(P) [→ M5]

BIG MOTOR: M10(P)→ M11(Q) [→ M7]

Additional loads may be identified. Board 4 was in fact not used as there were not enough loads to utilize the processors on it. MLM-16S provides the following load chains on boards 1, 2, and 3, as yet unused:

M0 (P)→ M1 (Q) [→ M5]

M3 (3P) [→ M5]

M4 (P)→ M6 (Q) [→ M5]

M10 (P) [→ M7]

M10 (P)→ M11 (Q) [→ M7]

Board 4 is configured to allow the use of the following load chains:

M12 (P)→ M13 (Q)→ M14 (3P) [→ M15]

M12 (P)→ M13 (Q) [→ M15]

M12 (P)→ M14 (3P) [→ M15]

M12 (P) [→ M15]

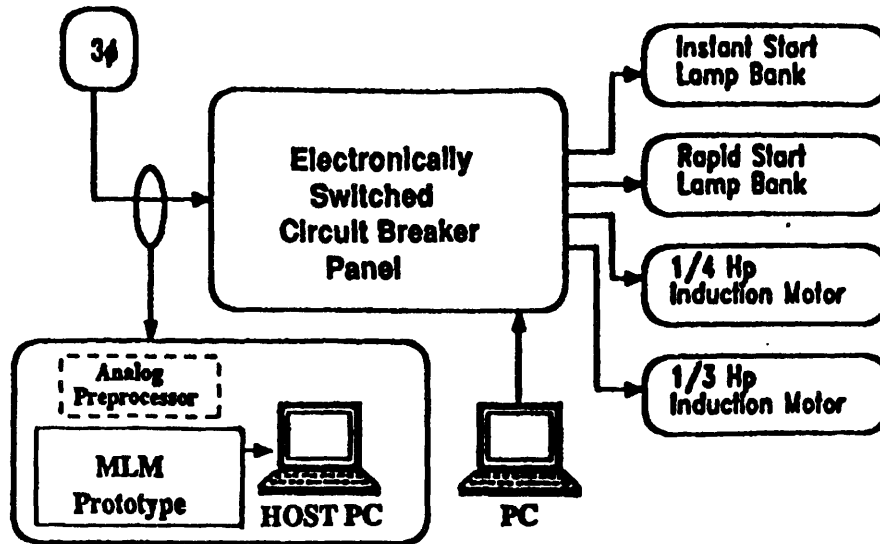


Figure 6.1: Prototype Test Facility

6.4 Results

The test stand used for experimentation was originally constructed as the prototype testing facility used in [1]. A block diagram of the stand is shown in Figure 6.1. It consists of the components of the platform for event detection, a collection of test loads (four of which are included in the figure), and an electronically switched circuit breaker panel which provided the electrical hookup to the loads. The monitoring platform consisted of the prototype MLM, the analog preprocessor and the host PC. A three phase electrical service powered the loads, which were chosen as representative of important load classes in medium to large size commercial and industrial buildings. In our experiments, the voltage and current waveforms of only one phase of the electrical service were monitored. The circuit breaker panel can support a total of eight devices: six single phase loads whose activities were monitored, and two three phase loads not included in our experiments. A dedicated computer controls the operation of each load connected to the circuit breaker panel through a collection of relays. This PC runs software that can turn the loads on and off in any sequence. It also allows the relative timing of the turn-on and turn-off events to be specified by the user. Hence, it can be programmed to start-up the loads to simulate a variety of possible end-use scenarios. It can ensure that turn-on events for different loads overlap — a feature used frequently in our experimentations.

The MLM monitors the electrical service entry of this “mock” building. It identifies the turn-on time and type of the loads activated. The MLM has, of course, no *a priori* knowledge of the operating schedule for the test loads. Experiments with this test facility are easily verified since the loads are activated under computer control. Moreover, these experiments are reasonably representative of the conditions that may be found on part of the wiring harness of an actual building [1].

We review here the results of using the MLM prototypes to monitor load activity. As mentioned above, a total of six loads were used in our experimentation. Results of 18 experiments are included here [the results given here are not “hand picked” or representative of specially tuned tests]. They range from a single load turning on to three or four devices turning on all at once. The following six loads’ activities were monitored:

1. 1/4 Hp Induction Motor
2. Bank of Rapid Start Fluorescent Lamps
3. Bank of Instant Start Fluorescent Lamps
4. Incandescent Light Bulbs
5. Computer
6. 1/3 Hp Induction Motor

MLM-8S was used to monitor loads 1, 2, and 3 while MLM-16S performed event detection for transients of all six loads. Results of detecting single load activity are given first. The two prototypes (in particular MLM-16S) were also tested with simultaneous multiple load start-ups, and these experiments are discussed next.

The results are displayed in Figures 6.2 through 6.19. These figures are the screen dumps of the plots of transient sections displayed by ECM (see Section 5.2). Recall from Chapter 5 that once the MLM is armed and the PC is in collater communication mode, it waits for a load turn-on event, and its subsequent identification, to occur. When this happens, identification data is uploaded from the collaters. During this result retrieval, the other slave processors go on acquiring and analyzing input data, so that the MLM never stops monitoring load activity. Several seconds of recently acquired data is also retrieved from the master board and displayed graphically. The displays show four plots consisting of the

envelopes of real power (P) in Watts, reactive power (Q) in VAR, and in-phase second (2P) and third (3P) harmonic contents of current. Loads identified as turning on are shown in the *MLM Console* Window along with the time of the event (see Section 5.2.9). As mentioned in Chapter 5, event detection was carried out on two time scales: Time scale 1 denotes the original (fine) scale and time scale 2 stands for the scale derived by downsampling the original data by 4 (the coarse scale). The scale (1 or 2) on which the load was detected is also reported. The following shorthand names were used to denote the loads monitored:

MOTOR = Small (1/4 Hp) Induction Motor

RAPID = Rapid Start Fluorescent Lamp Bank

INSTANT = Instant Start Fluorescent Lamp Bank

LIGHT = Incandescent Light Bulbs

COMP = Computer

BIG_MTR = Big (1/3 Hp) Induction Motor

Figures 6.2...6.7 show the results of the MLM's detection of solitary loads turning on. The graphs show the turn-on transients in the four streams for each event. The loads are seen as correctly identified. Figures 6.2 through 6.6 show the activity of loads detected on time scale 1. Figure 6.7 shows the captured turn-on transient for Big Motor. Note that the identification is made on time scale 2. The transient shapes shown in the first six figures should be studied and remembered. The v-sections chosen to represent each load are outlined in Appendix H. The v-sections for all loads on the original time scale were collected, in general, by walking through the transients and picking the edges that were seen as most repeatable. The templates for the big motor were not collected from raw input data. Instead, the small motor's v-sections were scaled in time and amplitude, and used for event detection on the coarse scale. The ease of template collection makes the performance of the MLM prototypes as load monitors, that much more remarkable.

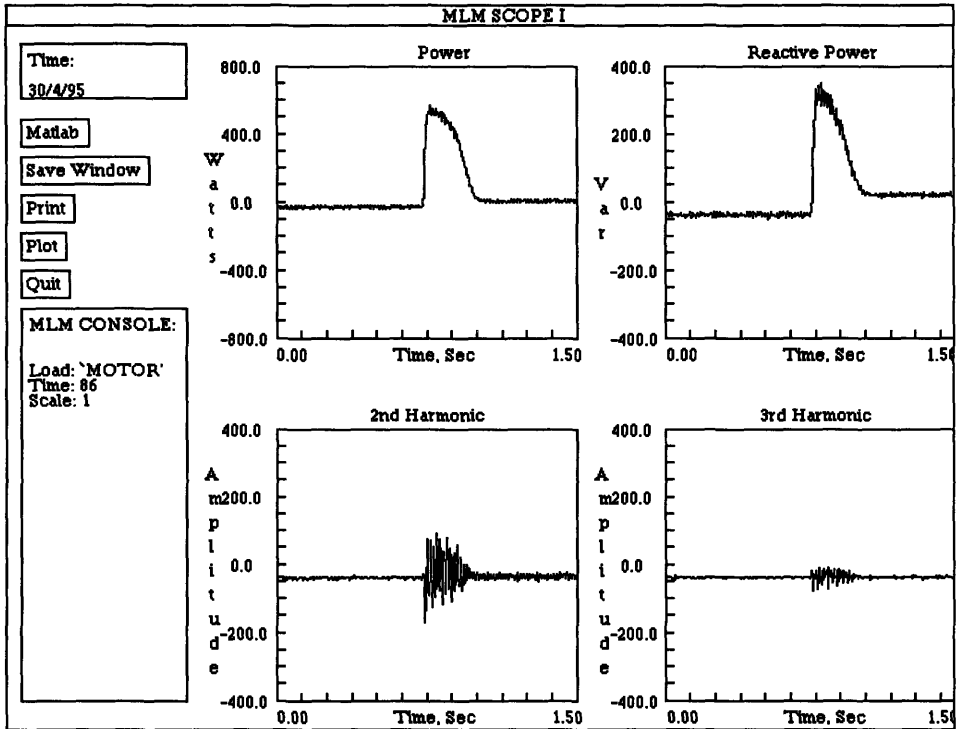


Figure 6.2: MLMscope Report: Small Motor

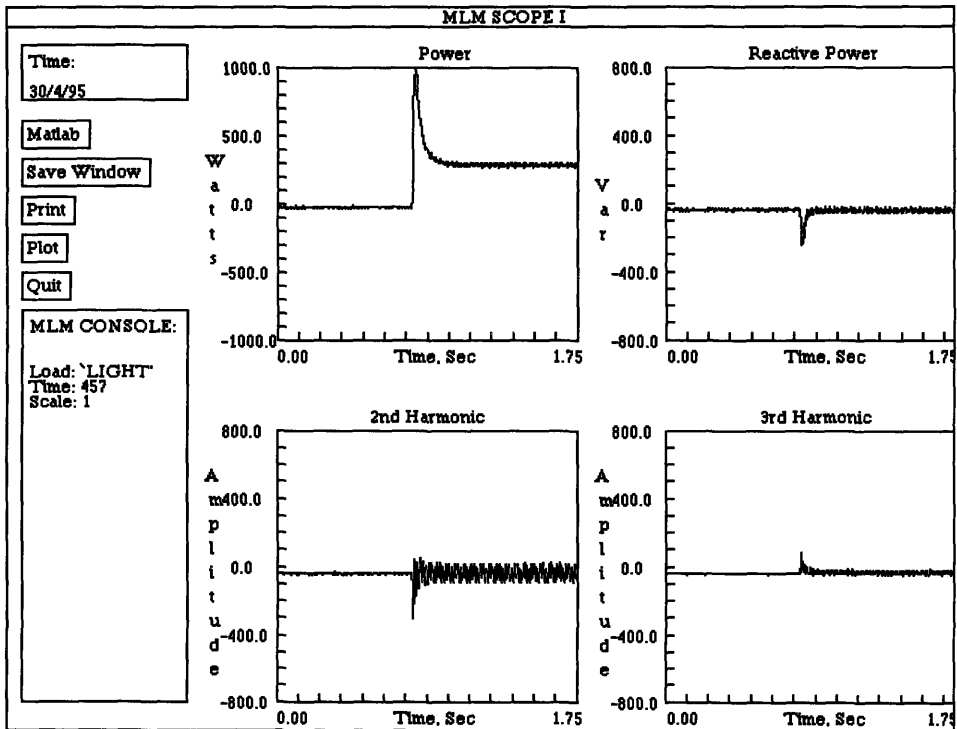


Figure 6.3: MLMscope Report: Incandescent Light Bulbs

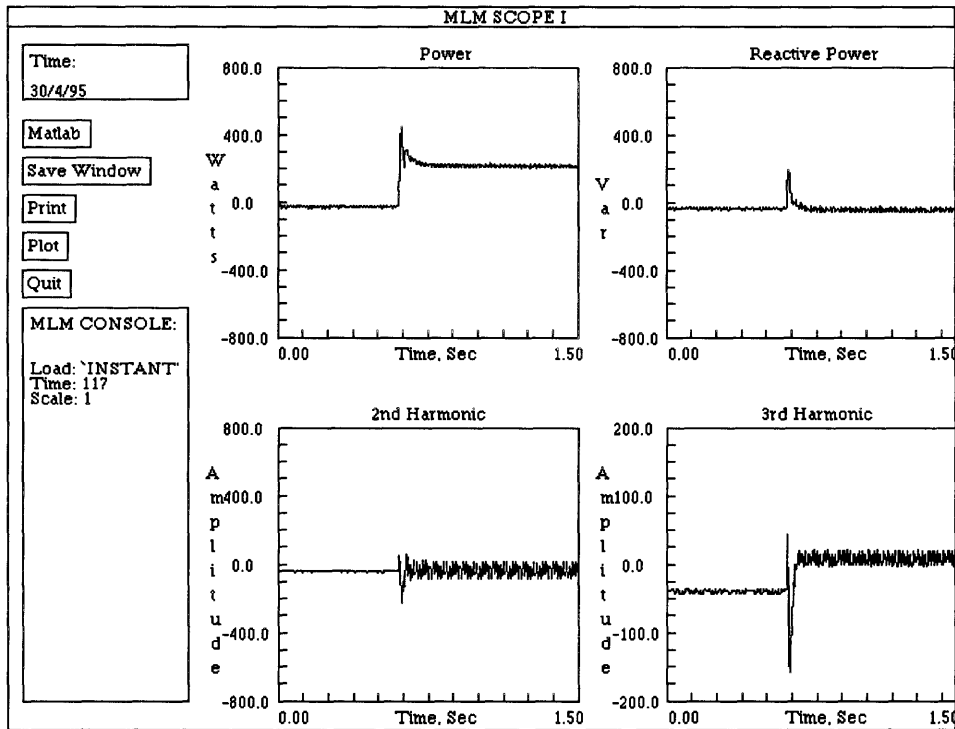


Figure 6.4: MLMscope Report: Instant Start Lamps

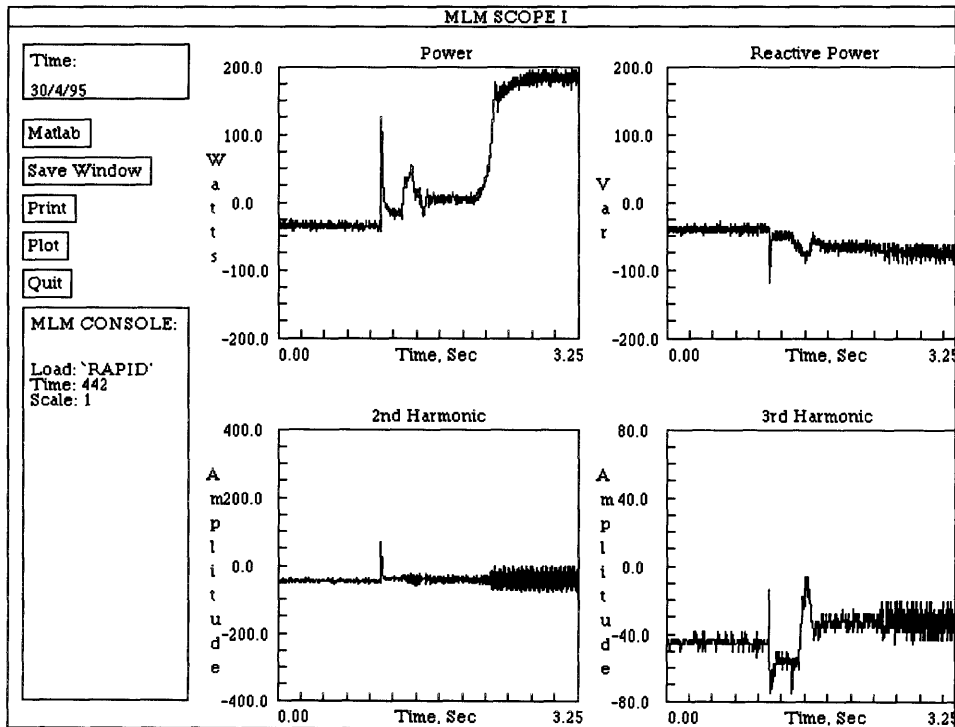


Figure 6.5: MLMscope Report: Rapid Start Lamps

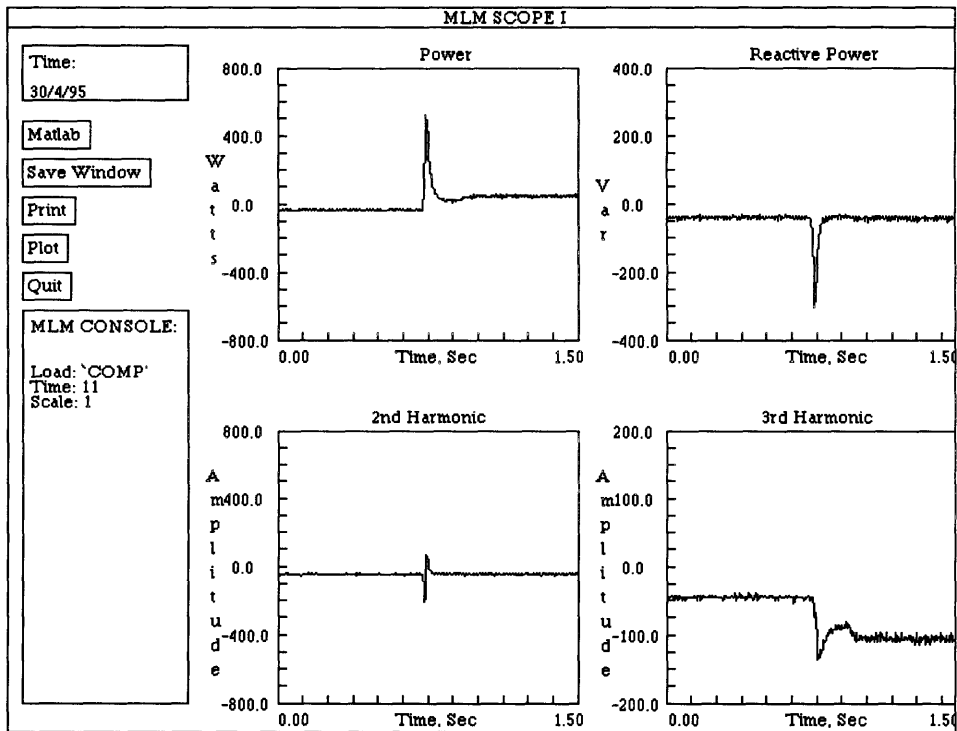


Figure 6.6: MLMscope Report: Computer

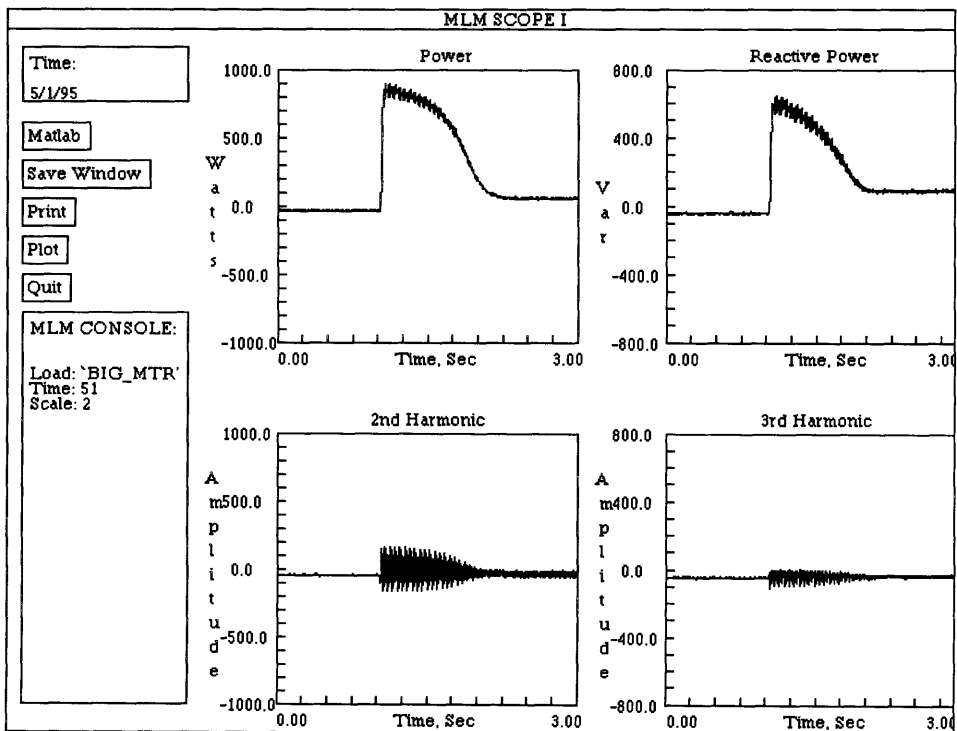


Figure 6.7: MLMscope Report: Big Motor

The transient shapes in Figures 6.2...6.7 should be studied carefully. This will help in examining the experiments with overlapping transients described below.

Figures 6.8 through 6.19 show the responses of the MLM when multiple loads were activated at about the same time. The transients overlap considerably, but the v-sections are distinct, and so the events are correctly identified. Table 6.5 gives a listing of these overlapping events as well as the delays used (in the software routines on the PC controlling the circuit breaker) to generate the sequence of load turn-ons in each case. For instance, entry 2 in Table 6.5 states that Figure 6.9 shows the result of turning on first the Rapid Start Lamps and, after a delay factor of 1550, the Small Motor. The term *delay* is a unitless count used to denote the software delay provided between consecutive load activations. See Appendix H for sample software routines used to activate multiple loads.

Refer to Figures 6.8 and 6.9 which show the small motor and instant start lamps being activated. In Figure 6.8, the motor is turned on first and, after a delay, the instant start lamps are started up. The v-sections do not overlap – indeed the entire main transients of the two loads do not overlap– so that load identification is easily achieved. Contrast this with the case in Figure 6.9, where the instant start lamps are started immediately after motor is turned on. The transients overlap completely. In fact, the main transients for the instant start lamps in P and Q occur plumb on top of the transients for motor. However the v-sections for the instant start lamps occur in quasistatic regions of the motor transients. Hence, all v-sections are recoverable, and the activity is correctly tracked.

Figures 6.10 through 6.12 show further instances of two loads turning on together and being detected on time scale 1. Figures 6.13 through 6.16 show experiments where three devices were turned on and identified on the fine time scale. In Figure 6.14, for instance, the rapid start lamps are turned on, followed by the instant start lamps and then the small motor. The transients in P, for example, for the instant start lamps and small motor occur in quasistatic sections of the transient for the rapid start lamps. While all three transients overlap completely, the v-sections are identified and the events are reported.

Figure 6.17 gives a good example of *multiscale* event detection, with the small motor correctly identified on time scale 1 and the big motor on scale 2. The big motor is turned on first, followed closely by the small motor. Note the complete overlap of transients in both P and Q. The v-sections are still recoverable and the load activity is thus detected correctly.

Figure	Loads Started (in order)	Delay 1	Delay 2	Delay 3
6.8	S. Motor, Instant	9500	–	–
6.9	S. Motor, Instant	1550	–	–
6.10	S. Motor, Light	1700	–	–
6.11	Rapid, S. Motor	12000	–	–
6.12	Rapid, Instant	13000	–	–
6.13	Rapid, S. Motor, Instant	12000	1550	–
6.14	Rapid, Instant, S. Motor	1550	10500	–
6.15	S. Motor, Light, Instant	1700	9000	–
6.16	Computer, S. Motor, Instant	10000	1550	–
6.17	B. Motor, S. Motor	7000	–	–
6.18	B. Motor, Rapid, S. Motor, Instant	30000	12000	1550
6.19	B. Motor, S. Motor, Rapid, Instant	7000	25000	1550

Table 6.5: Multiple Load Turn-on Delays

Finally, Figures 6.18 and 6.19 show four loads turning on: The transients overlap and event detection is carried out on two time scales. In Figure 6.18, the big motor turns on first, followed by the rapid start lamps, the small motor, and the instant start lamps. The transients of the latter three completely overlap, but all four loads are still correctly identified. In Figure 6.19, the small motor follows the start up of the big motor and the transients for the two overlap. This pair of transients is followed by the transients for the rapid start lamps and the instant start lamps, with the instant start lamps' transient occurring in the middle of the rapid start lamps' transient. All four loads are identified and the event records are shown in the MLM Console.

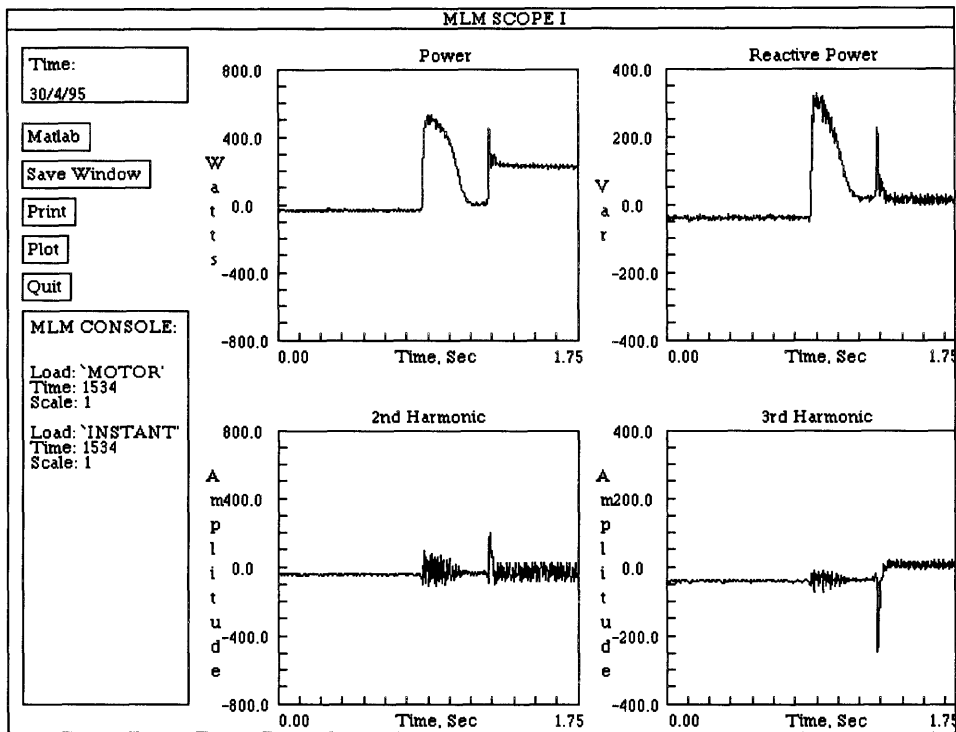


Figure 6.8: MLMscope Report: Small Motor, Instant

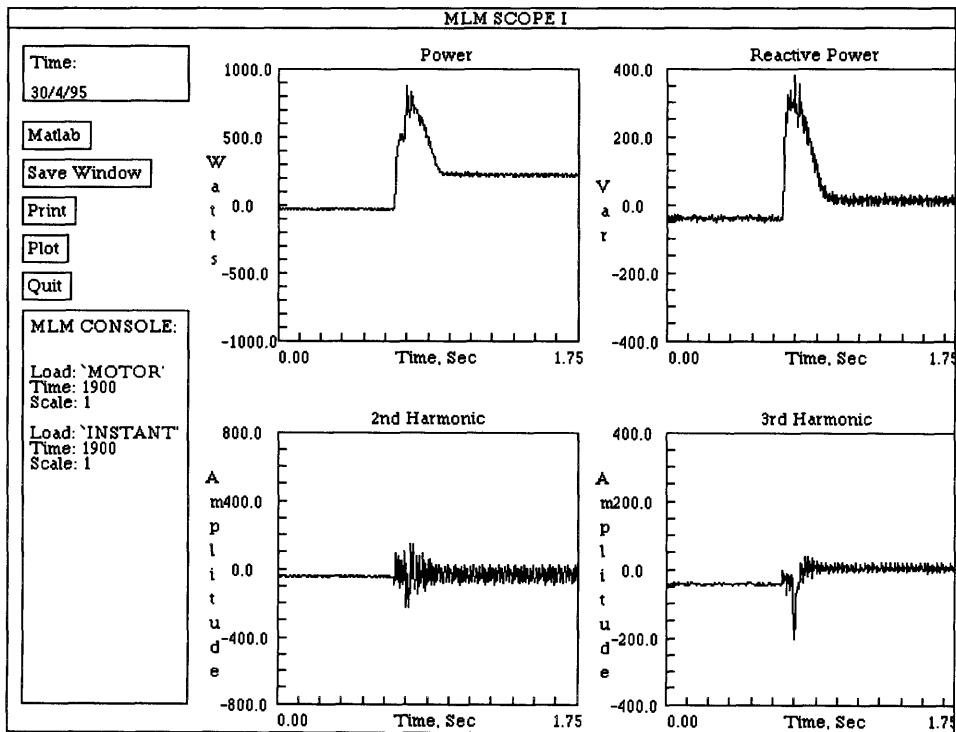


Figure 6.9: MLMscope Report: Small Motor, Instant

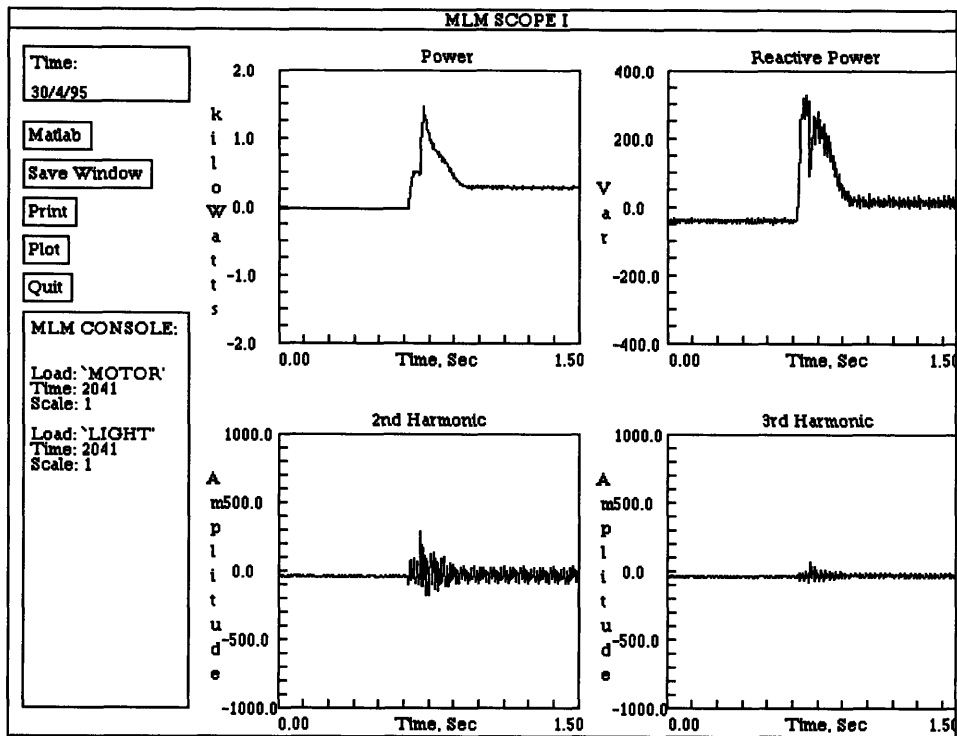


Figure 6.10: MLMscope Report: Small Motor, Light

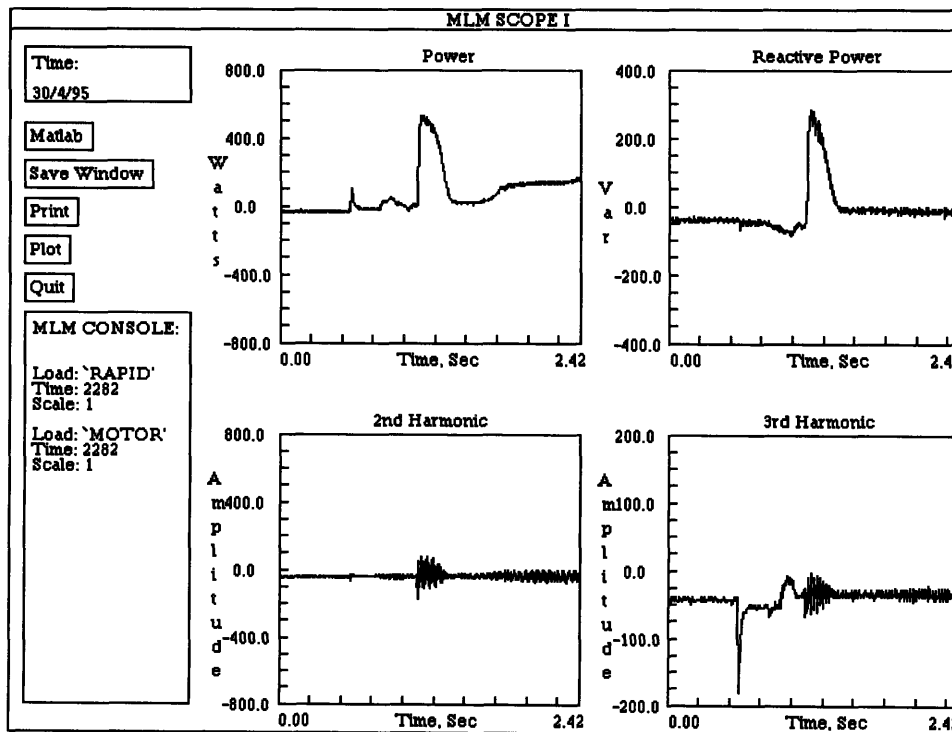


Figure 6.11: MLMscope Report: Rapid, Small Motor

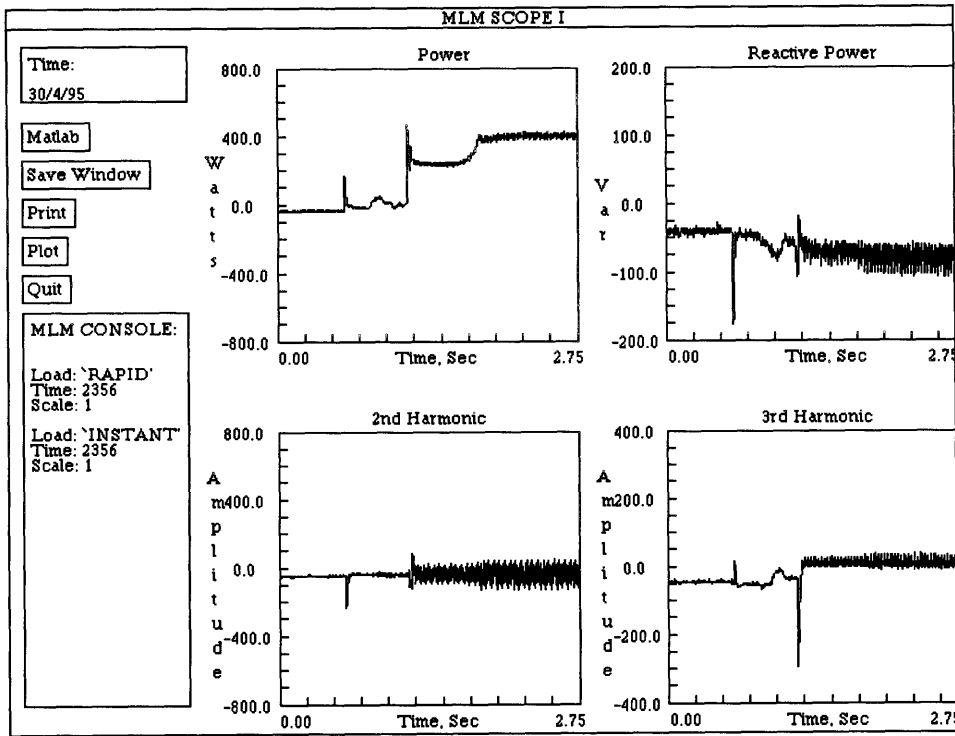


Figure 6.12: MLMscope Report: Rapid, Instant

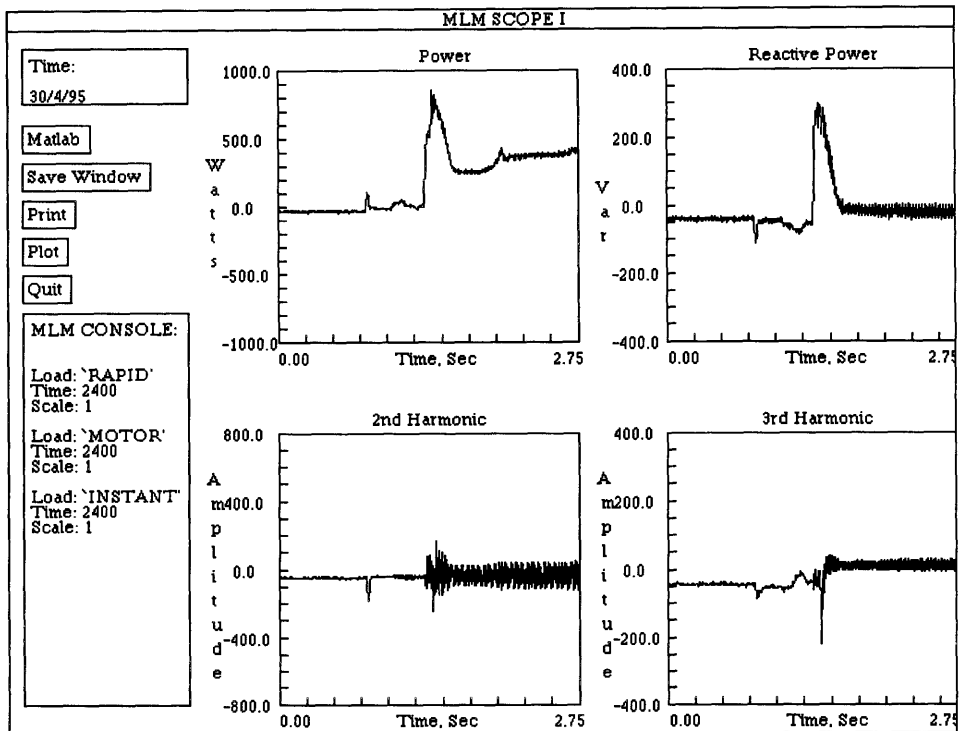


Figure 6.13: MLMscope Report: Rapid, Small Motor, Instant

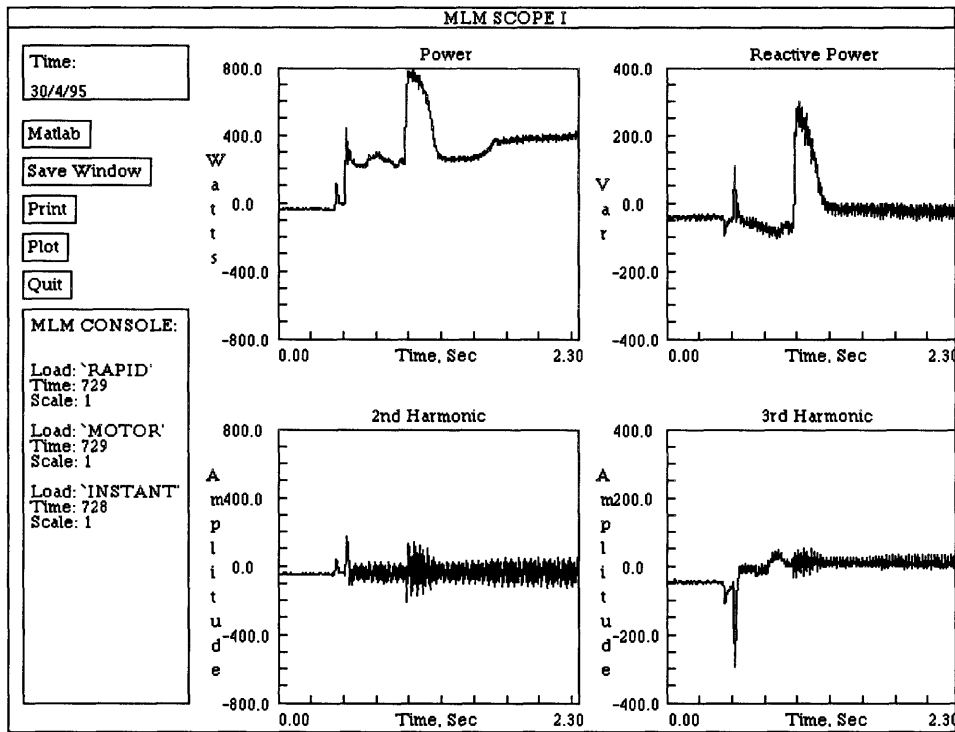


Figure 6.14: MLMscope Report: Rapid, Instant, Small Motor

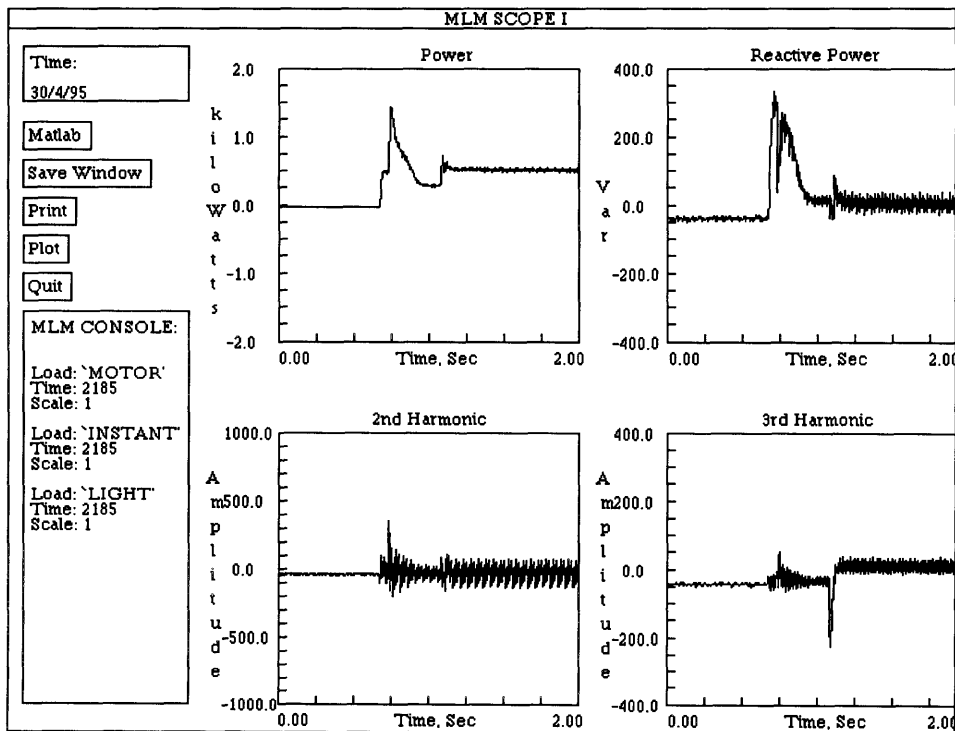


Figure 6.15: MLMscope Report: Small Motor, Light, Instant

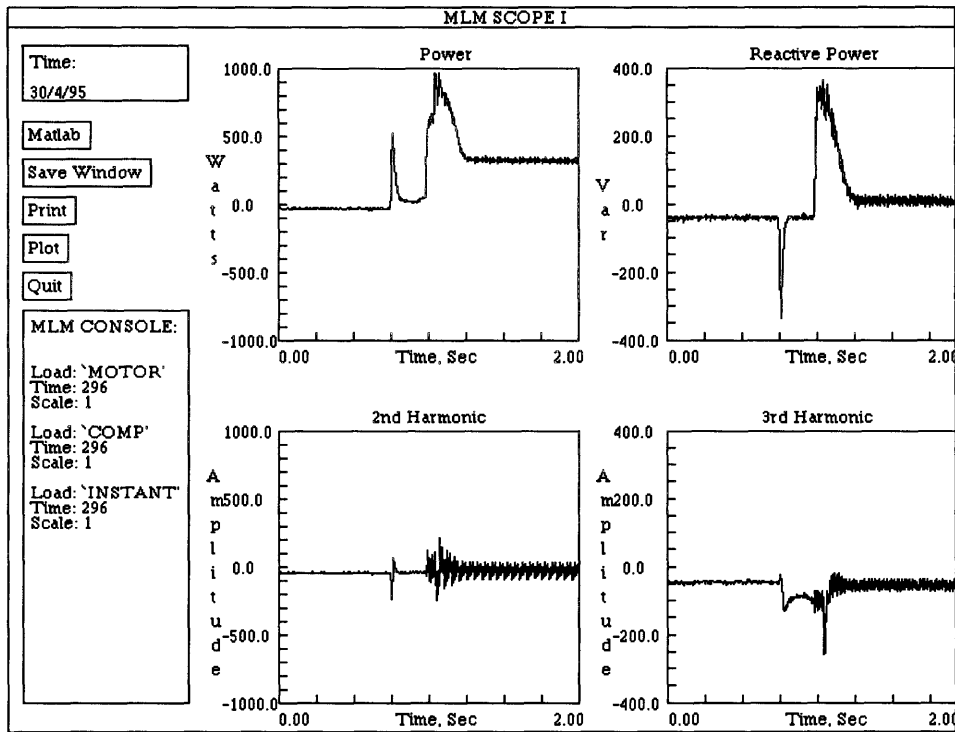


Figure 6.16: MLMscope Report: Computer, Small Motor, Instant

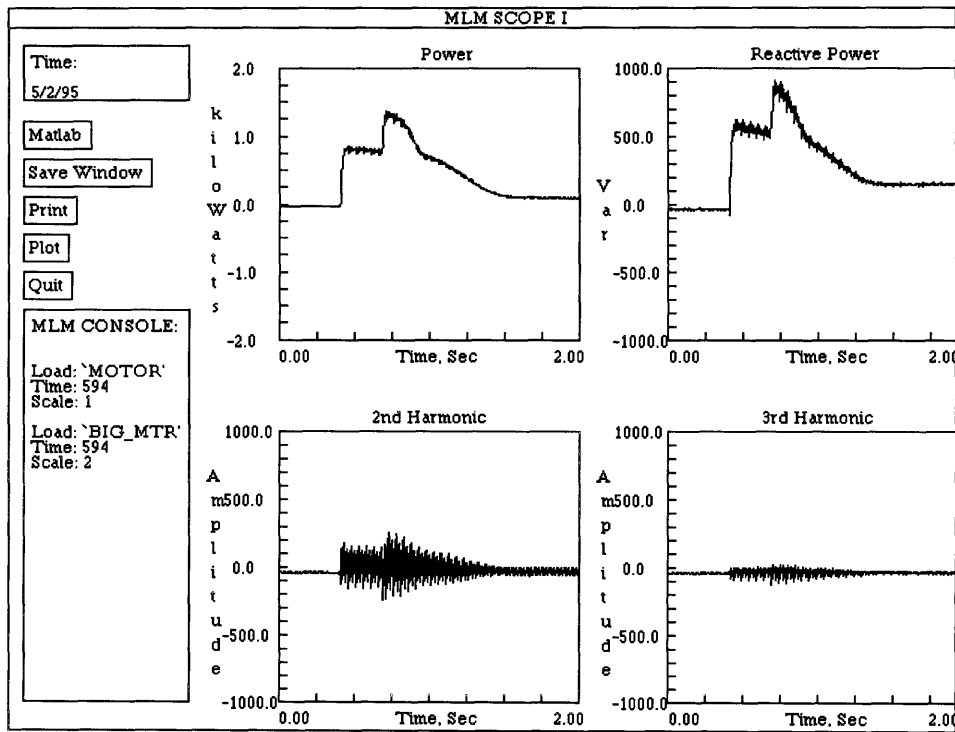


Figure 6.17: MLMscope Report: Big Motor, Small Motor

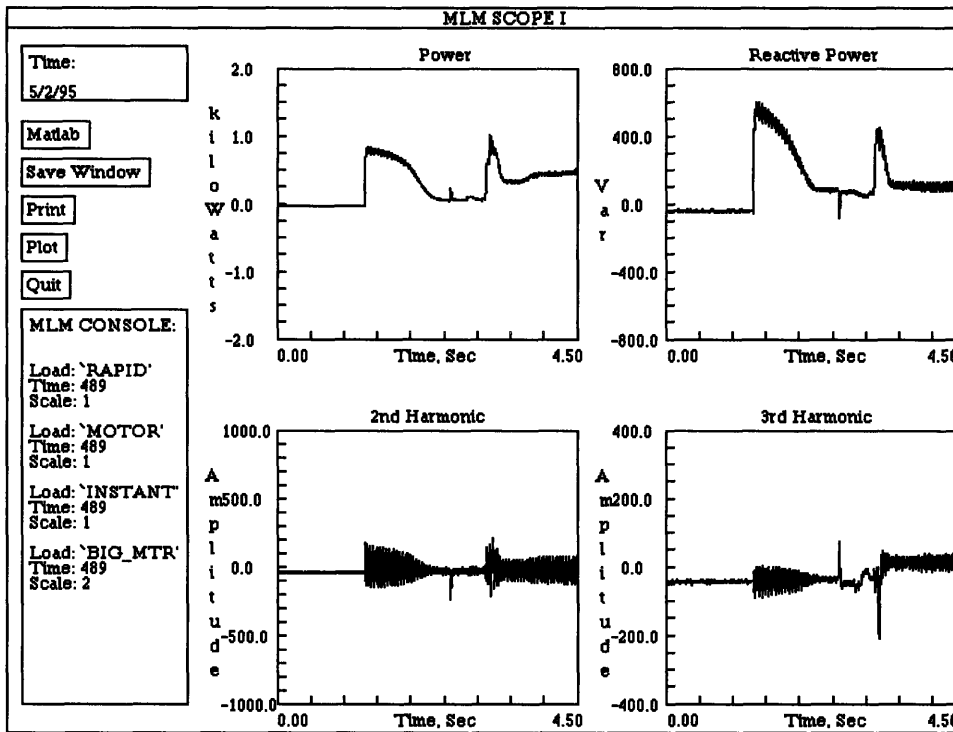


Figure 6.18: MLMscope Report: Big Motor, Rapid, Small Motor, Instant

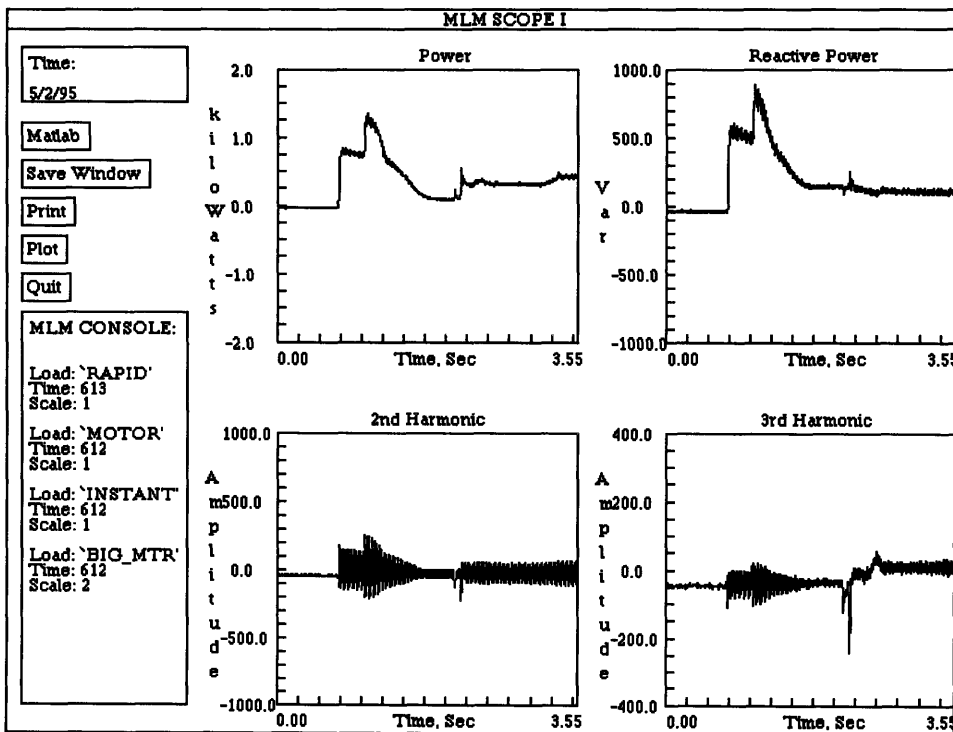


Figure 6.19: MLMscope Report: Big Motor, Small Motor, Rapid, Instant

6.5 Summary

Both MLM prototypes were consistently able to recognize the turn-on transients of the test loads and were not deceived into reporting false activity, unless of course the v-sections overlapped intractably. The versatility of the MLM was such that even seemingly intractable overlaps were processed correctly.

This performance is even more impressive considering the fact that relatively little effort was spent in collecting the v-sections and training the MLM on the loads (e.g. by adjusting error thresholds). The templates for all loads on the original time scale were collected by walking through the transients and picking the edges that were seen as most repeatable. The templates for the big motor were not, in fact, collected from raw data. Instead, the v-sections for the small motor were simply scaled in time and amplitude, and used by the slaves performing event detection on the coarse scale to identify the activity of the big motor. The fact that the MLM was still able to consistently identify the big motor gives credence to the idea of representing a class of loads by a single transient shape scaled in time and amplitude – a notion which can be fully utilized in *multiscale* transient event detection.

The loads tested showed much variation in the repeatability of their transients. Some, like the small motor, produced nearly identical transients time after time. Others showed annoying variations in their transient behavior. However, even with potentially troublesome loads such as the instant start lamps, the MLM did not falter. In the TED implementation in [1], the instant start lamps were a major problem to track, as the v-section in P is not reliable. Even after averaging several runs, a very dependable v-section was not achieved, so that the error threshold for identification in P had to be set high for this load. This could lead to false reports of load activity. With the MLM, however, the problem is solved by considering v-sections in several streams. Thus, in the case of the instant start lamps, the repeatable behavior in the 3rd harmonic saves the day. The instant start lamps are always identified and false hits are avoided as v-sections in both P *and* 3P must be found for an identification to be proclaimed.

Chapter 7

Conclusion

Summary of Results

We begin our recapitulation of the work done in this thesis by stating the premise of the research. Conventional nonintrusive load monitoring has relied on steady state behavior of loads. The research done in [1] pointed the way to a more versatile approach to determining the operating schedule of loads of interests, by considering their transient behavior. The multiscale event detector proposed in [1] advances the capabilities of the common NILM by using vector space techniques to identify transient sections. The goal of this research was to exploit the parallelism in the transient event detection (TED) algorithm and, having come up with a parallel approach to event detection, to implement the algorithm on a multiprocessing platform. By using an array of inexpensive processors interconnected in an intelligent manner, we could construct a potentially powerful and economically feasible platform for nonintrusive load monitoring. At the same time, fringe benefits such as power quality monitoring and future applications to diagnostic load evaluations could be developed. This was the discussion presented in Chapter 1.

Next, we developed a parallel version of the TED algorithm. In the context of the parallelized algorithm and complexity issues, an abstract model of the MLM was presented in Chapter 2.

In Chapters 3 and 4, functionality and design methodology of the main subsystems of the MLM, the **Master Board** and the **Slave Modules** are detailed. Their function and top-level design are described and supplemented with details of the actual implementation.

Chapter 5 rounds off our report on the design and implementation of the MLM with a

discussion of the system software. The code written for the computational units is described first, followed by the functions and software implementation of the Host PC User Interface.

Two prototypes were constructed for this thesis, one with 8 slave modules (MLM-8S) and the other with 16 processors (MLM-16S). Their final configuration is presented in Chapter 6. Results obtained during testing with real loads, and the performance achieved, are stated and evaluated.

The results obtained from the prototypes were extremely encouraging: the performance of the MLM was consistent and reliable. The versatility of the concept and the design was shown by correct load identification despite overlapping transients. The MLM verifies transient event detection as a valuable and reliable tool in load monitoring. The use of the inexpensive 80C196 microcontroller to achieve results comparable with, if not surpassing, the performance of an expensive DSP environment in executing the sequential algorithm consolidates the potency of a parallel implementation. The ease of scalability was demonstrated by the fact that construction of the bigger prototype required the simple process of beginning with one master and one slave board and adding on slave modules till the computational capacity needed was achieved. Thus, the performance of the MLM will always be expandable for more challenging load environments, by simply adding processors to identify more v-sections in various streams over several time scales. Such an expansion is not possible with a fixed sequential design.

The ratio of monitored loads per processors used was found to be quite reasonable. MLM-16S for instance has the capability of identifying between 14 to 18 loads with v-sections spanning four input streams and two time scales, giving a load per processor ratio of about 1.0. Note that the MLM-16S is in semi-custom configuration; i.e., while the loads to be identified were known, their exact characteristics (e.g., size and number of v-sections etc.) were not determined at the time of configuration. If the configuration had been customized for the number and types of v-sections after these had been studied, this ratio would have certainly increased. Moreover, if a v-section search is conducted on one time scale so that the cost of time-scaling each stream and collating separately on each time scale is eliminated, the load/processor ratio can be increased further. This is good news if we need to monitor a group of loads on the same transient time base. With the future work that may be done on this platform, and with the availability of cheaper, more versatile processors, (as discussed in the next section), this ratio could be significantly increased.

Power quality monitoring is also possible using the MLM built in this thesis. At present the MLM acquires and makes available to the user the 2nd, 3rd, and 5th harmonics of current. Together with the load identification capability of the MLM, this information would make it possible to easily track down power quality offenders even in a busy commercial environment.

Directions for Future Work

The MLM's performance underscores the versatility and power of conducting multiscale transient event detection using an application-specific multiprocessing computer. Nevertheless, there remain several areas in design and implementation that require future work before a commercially feasible nonintrusive load monitor is produced.

One process that needs refinement is the manual collection of v-section templates. As the number of loads to be monitored increases or as the number of v-sections required per load is increased, extracting templates manually, even with advanced MATLAB features at one's disposal, is going to be a daunting task. Automating template collection would be a much-needed improvement in the MLM user interface. First of all, this would require the automation of the process by which ECM fetches data from the master board following load activity. Next, signal processing would be performed on the data, probably in MATLAB, or an equivalent application, to isolate v-sections. Once v-sections are identified, they would be placed in separate files in the proper format. Finally, the data would be ac-coupled by using either a DOS batch file to invoke ECM to perform this task, or writing out separate code for the process.

As mentioned in Chapter 2, there is considerable research to be done in coming up with the optimal configuration of the MLM. Multiscale transient event detection is accomplished on the MLM by distributing the v-section pattern search, tree-structured decomposition, and collation, over several computational units. In addition, the v-sections associated with the loads must be assigned specific computational units. An important direction for future work would be to analytically determine the optimal distribution of these various functions and load transient assignments over the computational units. The MLM is designed in fact, to provide an experimental platform to test theories of optimal implementation of the multiscale event detector algorithm in a parallel processing environment.

The next step towards an economically viable load monitor would be to streamline the design of the MLM for cost. Several immediate steps can be taken: the boards may be implemented on 4–6 layer PCBs instead of the 2-layer boards in the current version; the on-chip ROM version of the 80C196 could be used; surface mount technology could be used to reduce PCB real estate cost, as well as the overall bulk of the load monitor; a smaller, cheaper external UART could be incorporated in the slave modules in place of the I82510. More involved options include moving to a new architecture that offers the same computational prowess for lower cost.

Increasing the performance of the MLM as separate from optimizing for cost, would be another interesting avenue for future work. One important starting point is to remove the bottle-neck created by the limited HSI/O lines available for load chaining. Potentially, v-sections belonging to more than four loads may be handled by a processor. However, the number of chains passing through an *intermediate* processor or an *end* processor is restricted to two. The slave that is the *first* processor for all its associated load chains has a limit of four chains imposed on it. At present, no serial communication occurs between processors on a chain. This stifles the flexibility of load assignment; in particular this implies that all v-sections on a time scale for a load, on one data-stream, must reside in a single processor. Till now these limitations have not proven problematic. However, as the challenges in load monitoring increase and as the basic computational unit used becomes more efficient, these design issues would need to be dealt with.

The MLM is a potentially valuable platform for nonintrusive diagnostic evaluations of industrial loads. The MLM allows raw data to be shipped — in bulk — directly to the host PC, while reporting all events detected. This wealth of data could be harnessed by the proper techniques and algorithms, to perform pre-emptive diagnosis. Results reported in [26] and [27] show how multirate estimators can be used to determine the state space parameters for induction motors using only measurements made at the electrical terminals. It is conceivable that similar techniques could be developed for other types of rotating electric machinery, and for other loads. By tracking trends in parameters over a period of time, it may be possible to say something about the “health” of the monitored loads. The capability to foretell the need for maintenance would make the MLM invaluable in an industrial or commercial environment.

The work of this thesis is a major step towards the construction of a powerful and

economically viable nonintrusive load monitor for commercial and industrial environments.

Bibliography

- [1] S.B. Leeb, "A Conjoint Pattern Recognition Approach to Nonintrusive Load Monitoring," M.I.T. Ph.D. Thesis, Department Of Electrical Engineering and Computer Science, February 1993.
- [2] *Power Pollution Caused by Fluorescent Lighting Fixtures*, Power Quality Laboratory TechBrief, California Polytechnic, San Luis Obispo, Vol 1., No. 12, September 1990.
- [3] R. Tabors, et. al., "Residential Nonintrusive Appliance Load Monitor," EPRI Final Report, to appear.
- [4] B. Wilkenson, "Power Factor Correction and IEC 555-2," *Powertechnics Magazine*, February 1991, pp. 20-24.
- [5] R.O. Duda and P.E. Hart, *Pattern Classification and Scene Analysis*, John-Wiley and Sons, 1973.
- [6] S.A. Ward and R.H. Halstead, *Computation Structures*, The MIT Press, 1990.
- [7] J.L. Hennessy, D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers Inc., 1990.
- [8] S.B. Leeb, S.R. Shaw "Harmonic Estimates for Transient Event Detection," Universities Power Engineering Conference (UPEC), Galway, Ireland, 1994.
- [9] Analog Devices, *Data Converter Reference Manual, Volume II*, 1992.
- [10] Integrated Device Technology Inc., *Fast Static RAM databook*, 1993.
- [11] CAD Software Inc., *PADS-PCB Reference Manual: Volume I*, 1988.
- [12] CAD Software Inc., *PADS-PCB Reference Manual: Volume II*, 1988.

- [13] Intel Corporation, *80C196KC: User's Manual*, 1992.
- [14] Intel Corporation, *EV80C196KC: User's Manual*, 1992.
- [15] Intel Corporation, *Microcommunications*, 1992.
- [16] Intel Corporation, *Microprocessors and Peripheral Handbook, Volume II*, 1990.
- [17] Intel Corporation, *IC-96 C Compiler for 80C196*, 1992.
- [18] B.W. Kernigan and D.M. Ritchie, *C Programming*, 2nd edition, Prentice Hall, 1988.
- [19] A.V. Oppenheim, R.W. Schaffer, *Discrete-Time Signal Processing*, Prentice Hall, 1989.
- [20] W.M. Siebert *Circuits, Signals and Systems*, The MIT Press, 1986.
- [21] Microsoft Corporation, *Microsoft C Compiler, Version 6.0: User's Guide*, 1990.
- [22] Intel Corporation, *ASM-96 Assembler for 80C196*, 1992.
- [23] JDR Microdevices, *JDR PR-1 and PR-2 Users Manual*, 1986.
- [24] M. Sargent III, R.L. Shoemaker, *The IBM PC From the Inside Out*, revised edition, Addison-Wesley Pub. Co., 1986.
- [25] P. Norton, *Inside the IBM PC*, Brady, New York, 1986.
- [26] M. Veles-Reyes, "Speed and Parameter Estimation for Induction Machines," M.I.T. S.M. Thesis, Department Of Electrical Engineering and Computer Science, May 1988.
- [27] M. Veles-Reyes, K. Minami, G. Verghese, "Recursive Speed and Parameter Estimation for Induction Machines," *Proceedings of IEEE Industry Applications Society Annual Meeting*, 1989, pp. 607–611.
- [28] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

Appendix A

Master Board Schematics and Layouts

The following schematics and plots are included in this appendix:

- Figure A.1: Master Board Schematic 1.
- Figure A.2: Master Board Schematic 2.
- Figure A.3: Master Board Schematic 3.
- Figure A.4: Master Board Schematic 4.
- Figure A.5: Master Board Schematic 5.
- Figure A.6: PCB Component Layout for Master Board.
- Figure A.7: PCB Component and Solder Side for Master Board.

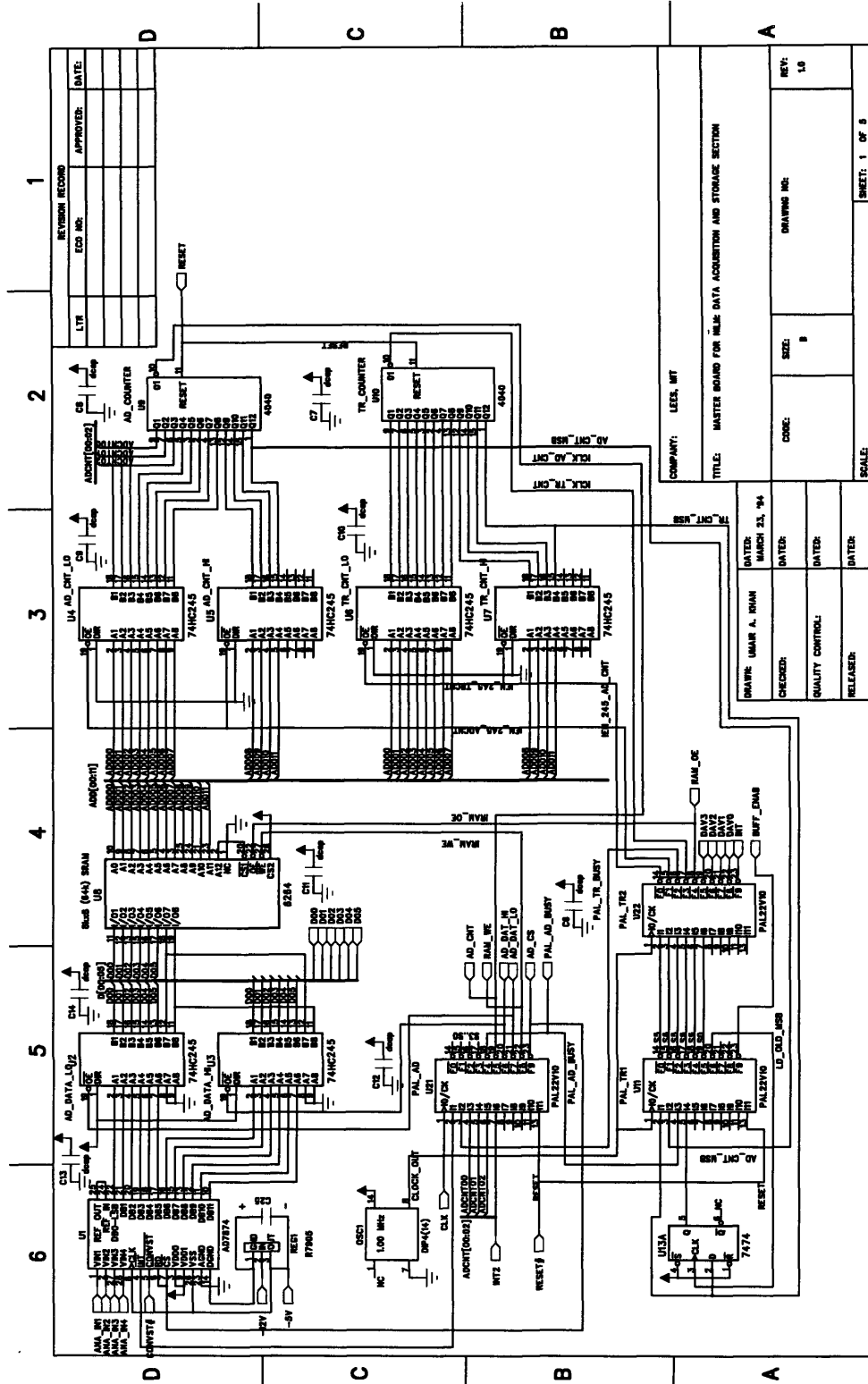


Figure A.1: Master Board Schematic 1

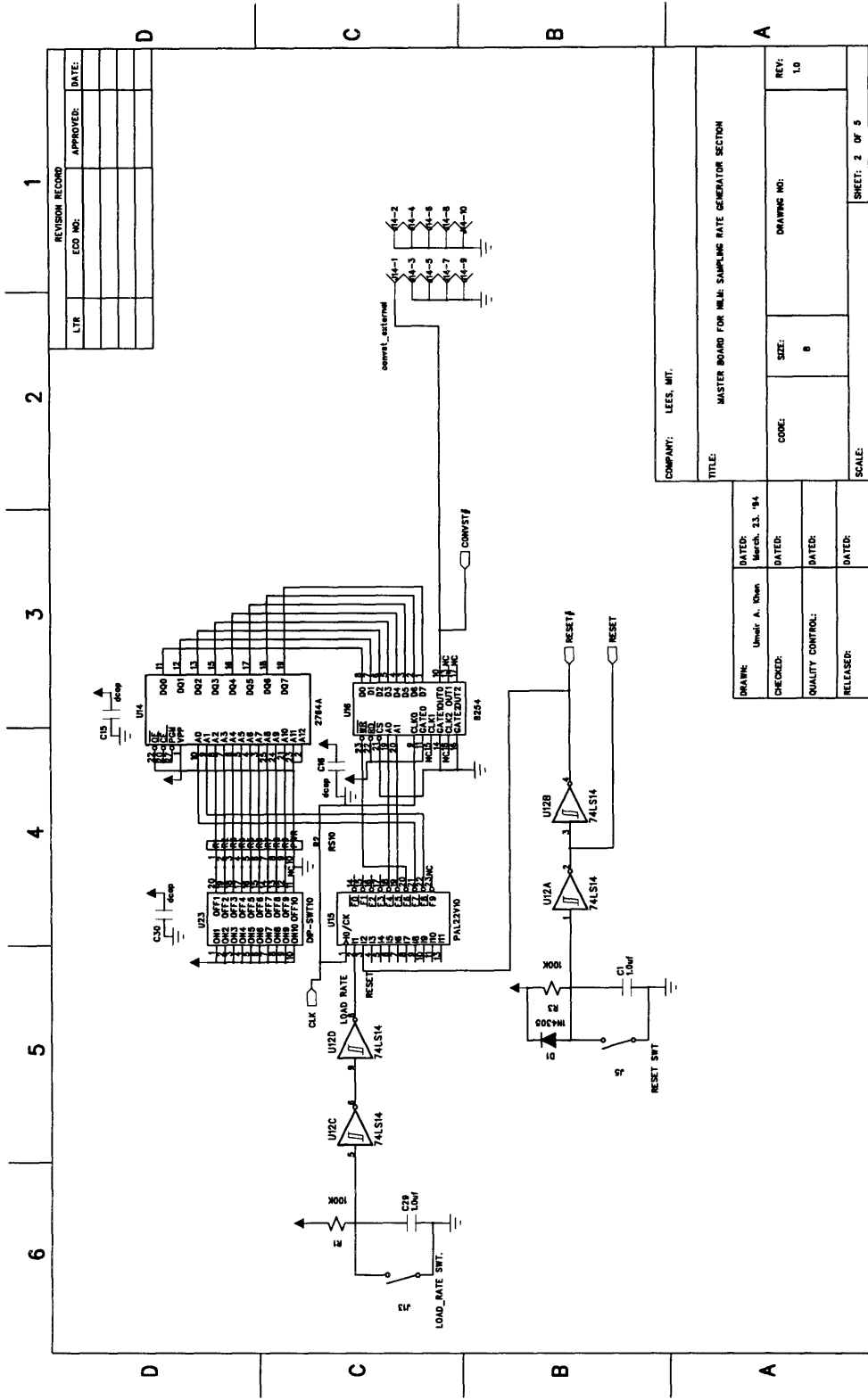


Figure A.2: Master Board Schematic 2

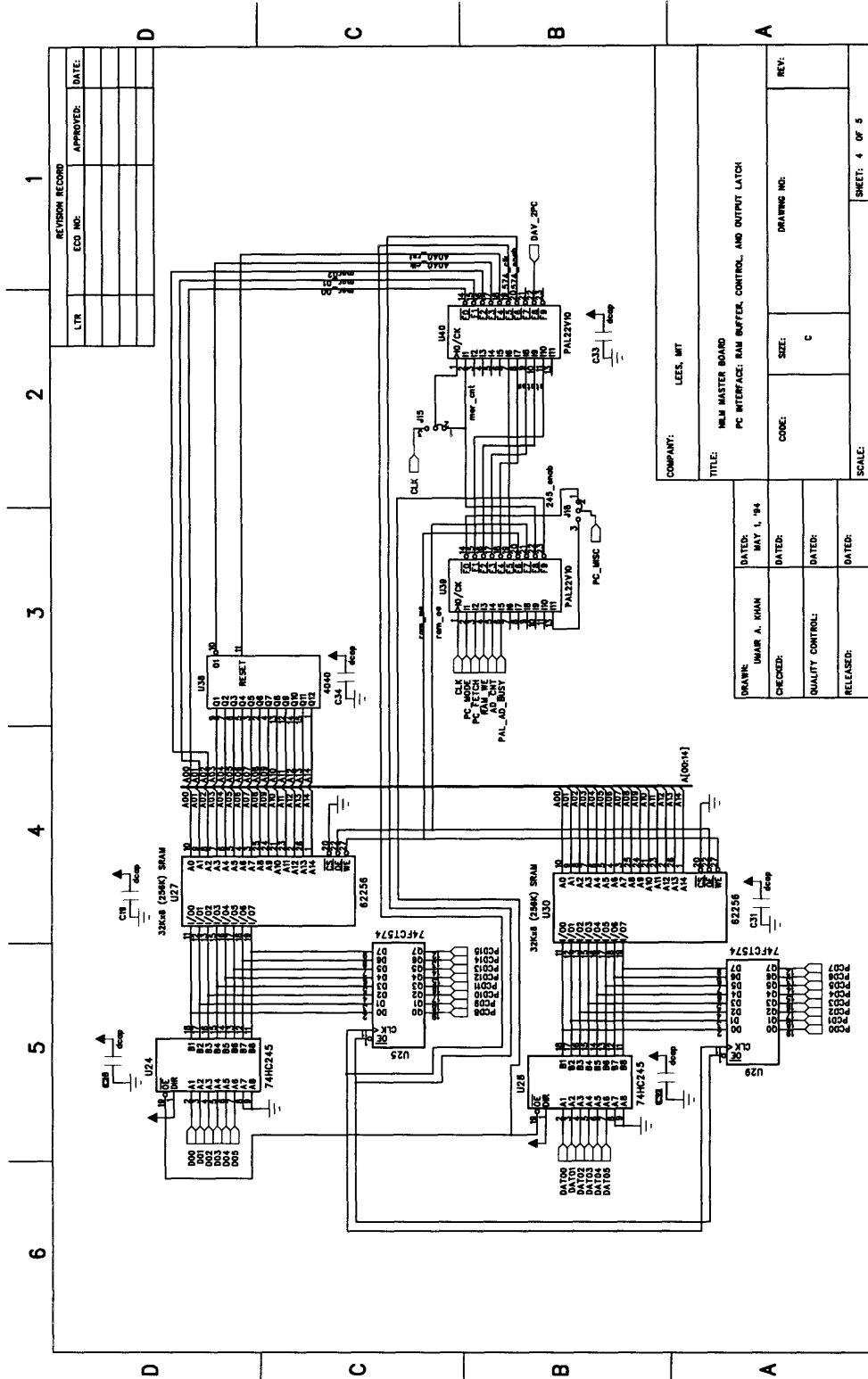


Figure A.4: Master Board Schematic 4

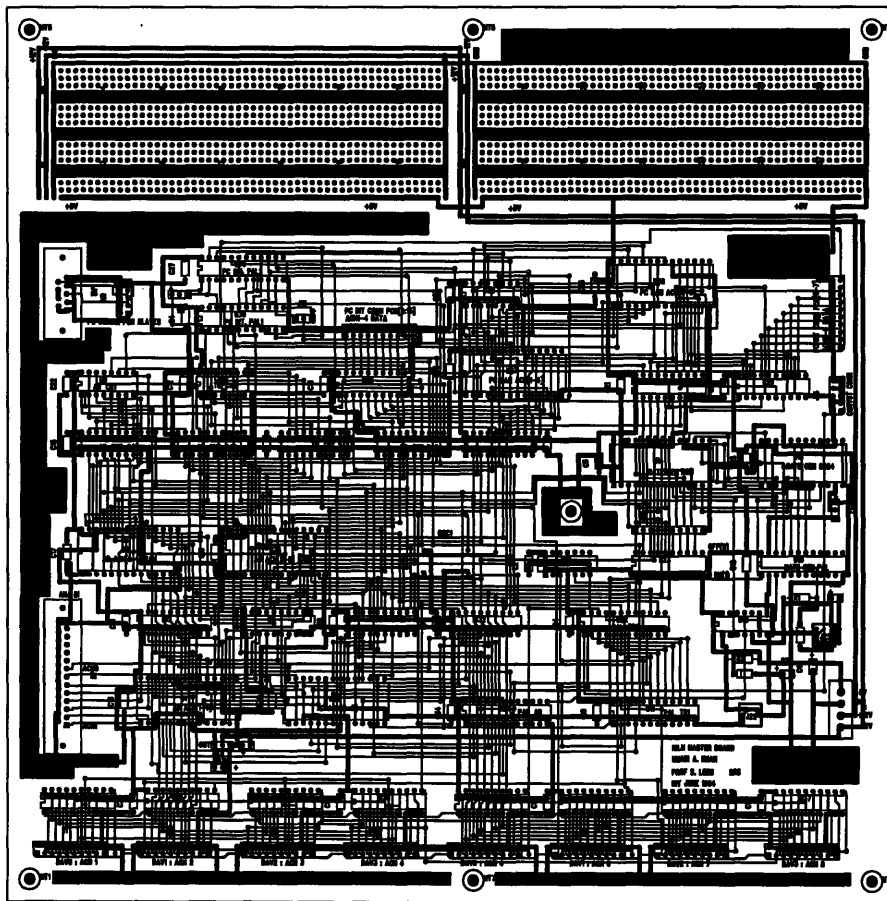


Figure A.7: Master Board: PCB Component and Solder Sides

Appendix B

Slave Board Schematics and Layouts

The following schematics and plots are included in this appendix:

- Figure B.1: Slave Board Schematic 1.
- Figure B.2: Slave Board Schematic 2.
- Figure B.3: Slave Board Schematic 3.
- Figure B.4: Slave Board Schematic 4.
- Figure B.5: Slave Board Schematic 5.
- Figure B.6: PCB Component Layout for Slave Board.
- Figure B.7: PCB Component and Solder Sides for Slave Board.

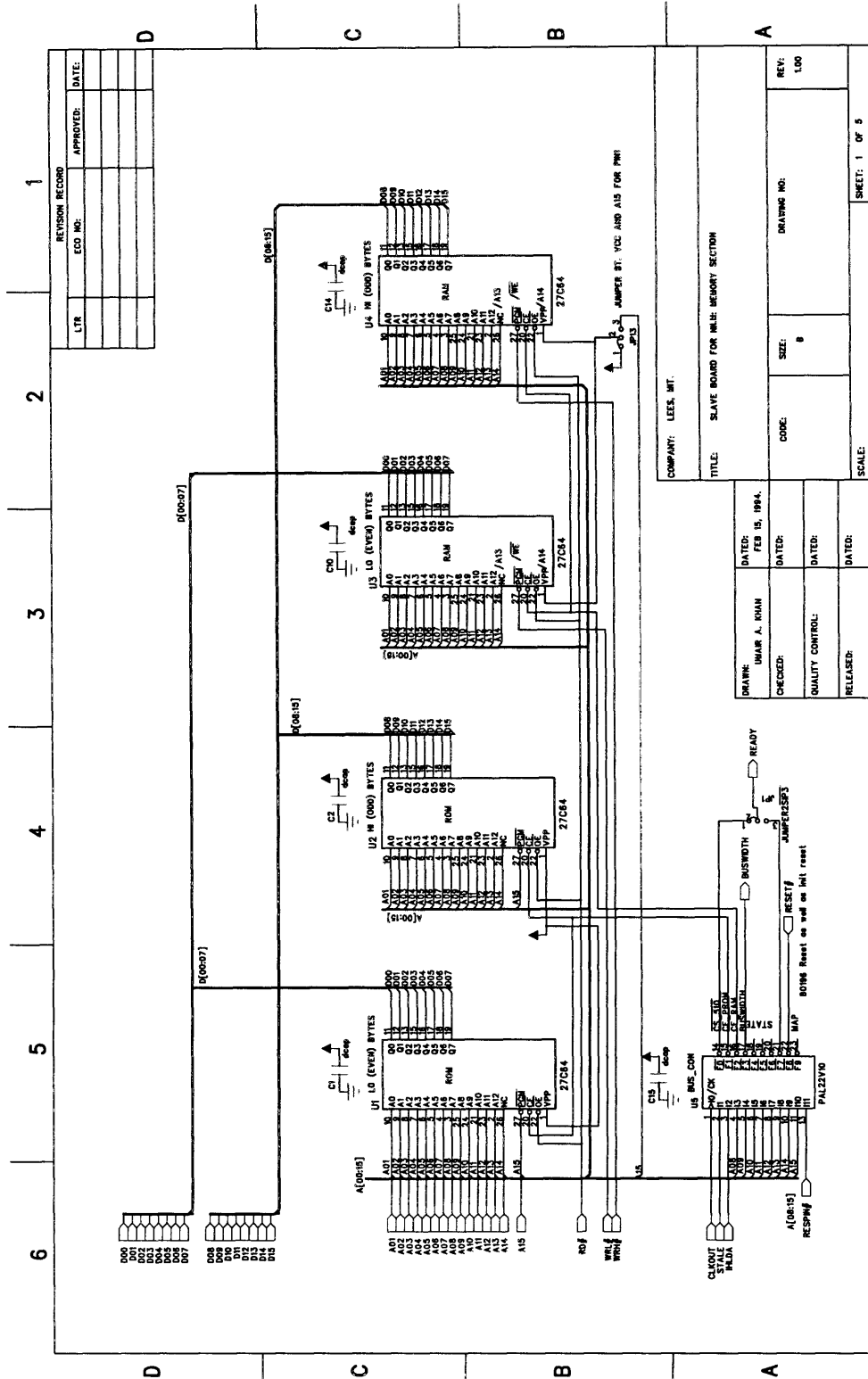
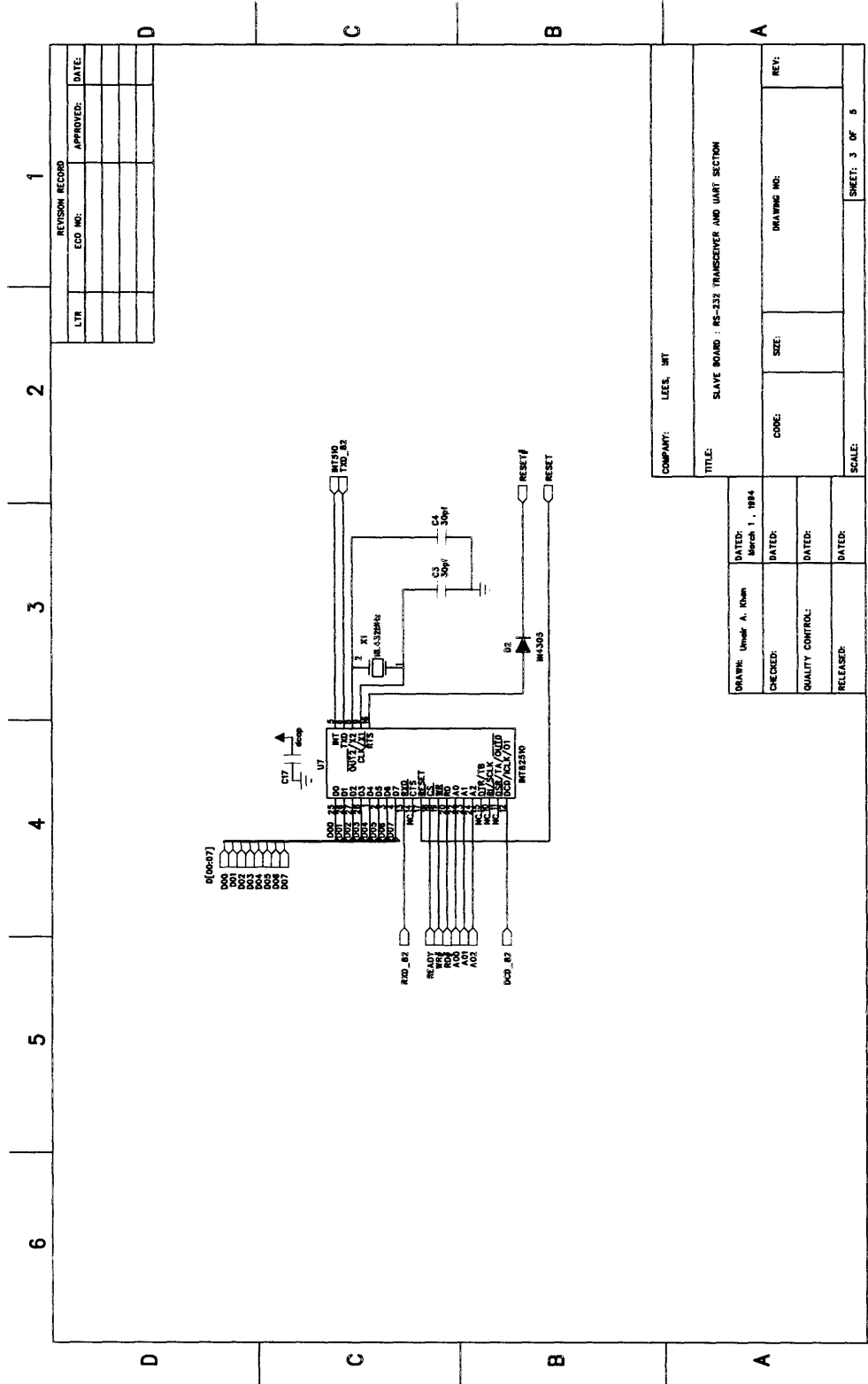


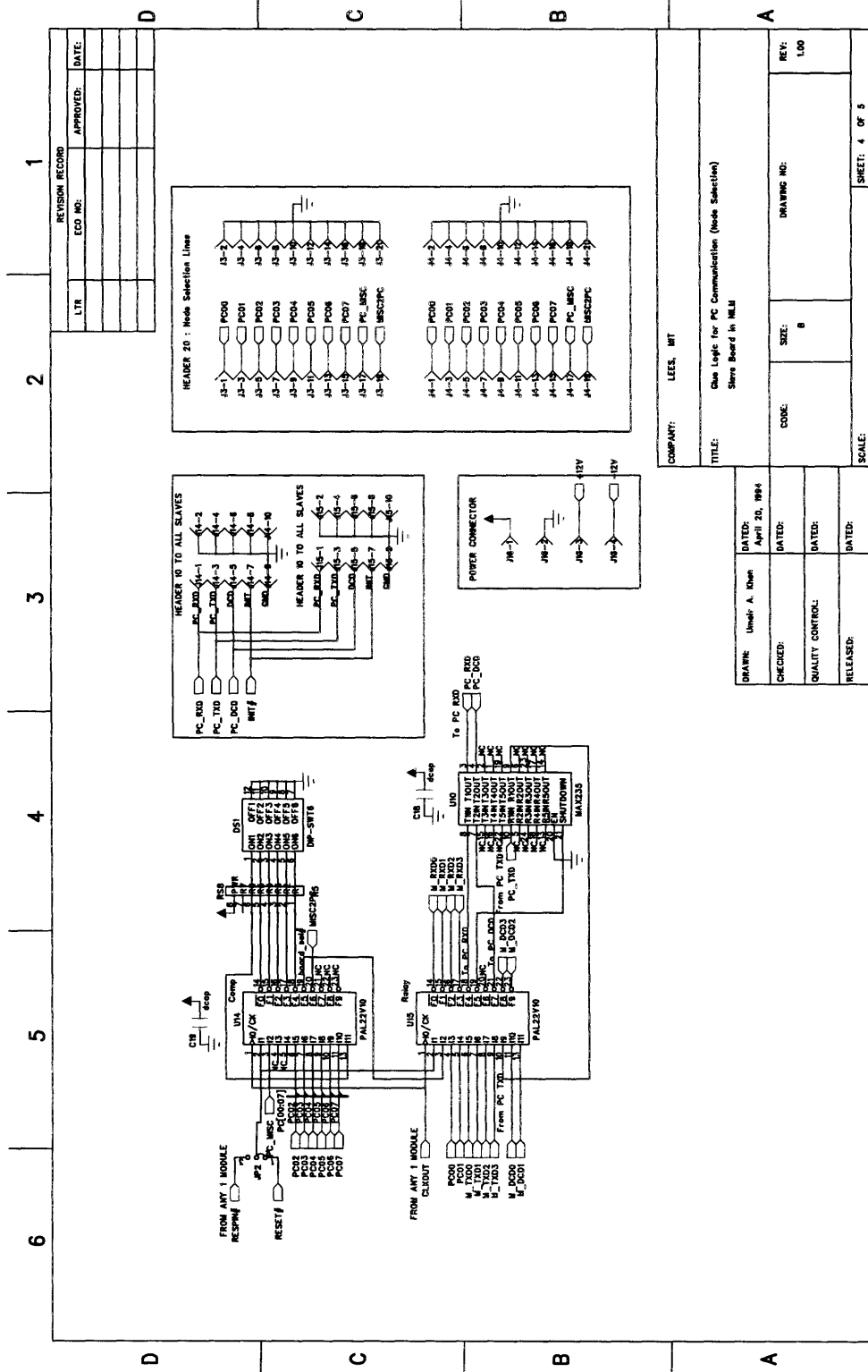
Figure B.1: Slave Board Schematic 1



REVISION RECORD		
LTR	ECO NO:	DATE:

COMPANY: LEES, INT	
TITLE: SLAVE BOARD : RS-232 TRANSMITTER AND UART SECTION	
DRAWN: Umehr A. Khen	DATED: March 1, 1984
CHECKED:	DATED:
QUALITY CONTROL:	DATED:
RELEASED:	DATED:
CODE:	SIZE:
DRAWING NO:	REV:
SCALE:	SHEET: 3 OF 5

Figure B.3: Slave Board Schematic 3



REVISION RECORD	
LTR	DATE:
	APPROVED:
	ECO NO.:

COMPANY: LEES, MT	
TITLE: One Logic for PC Communication (Node Selection) Slave Board in HLI	
DATE: April 20, 1984	DATE:
CHECKED:	DATE:
QUALITY CONTROL:	DATE:
RELEASED:	DATE:
CODE:	SIZE: B
DRAWING NO.:	REV: 100
SCALE: SHEET: 4 OF 5	

Figure B.4: Slave Board Schematic 4

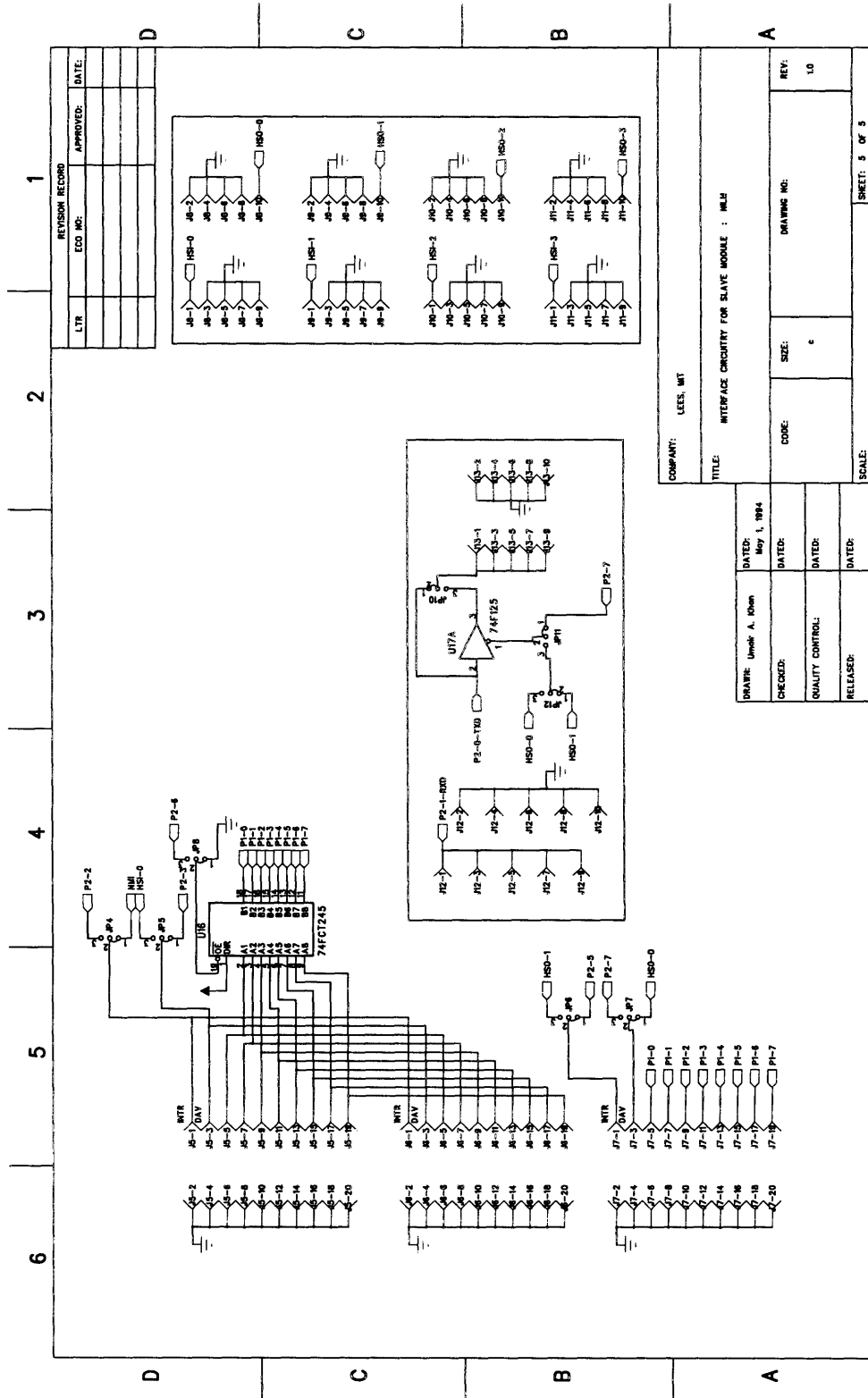


Figure B.5: Slave Board Schematic 5

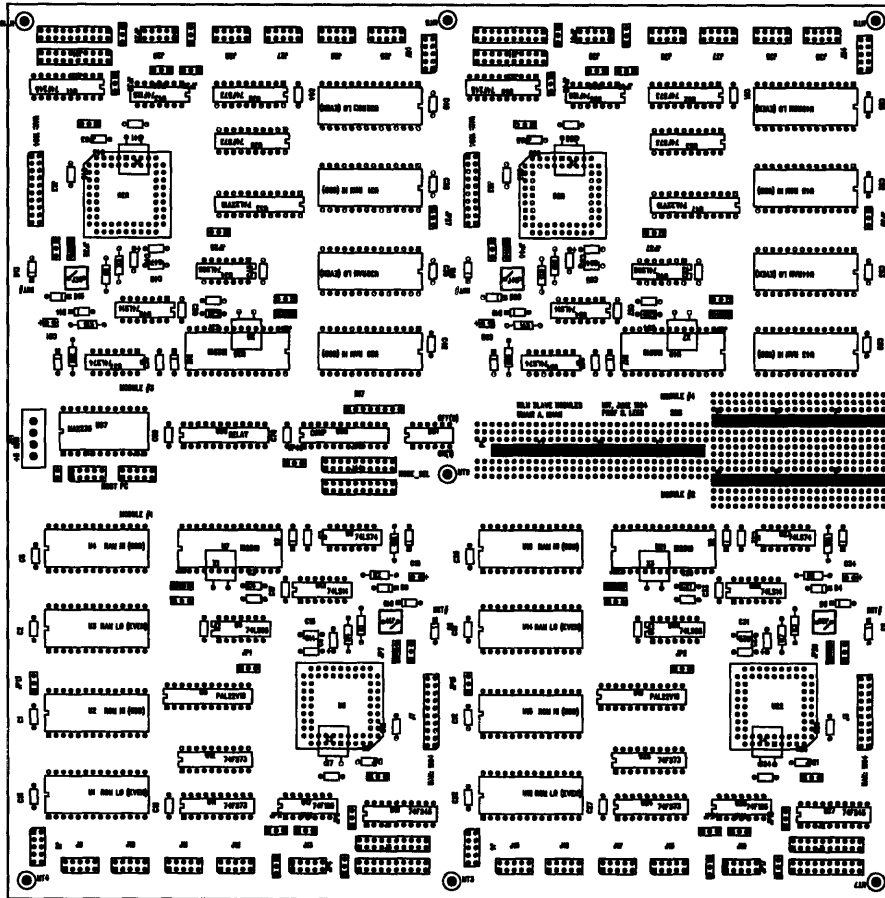


Figure B.6: Slave Board: PCB Component Layout

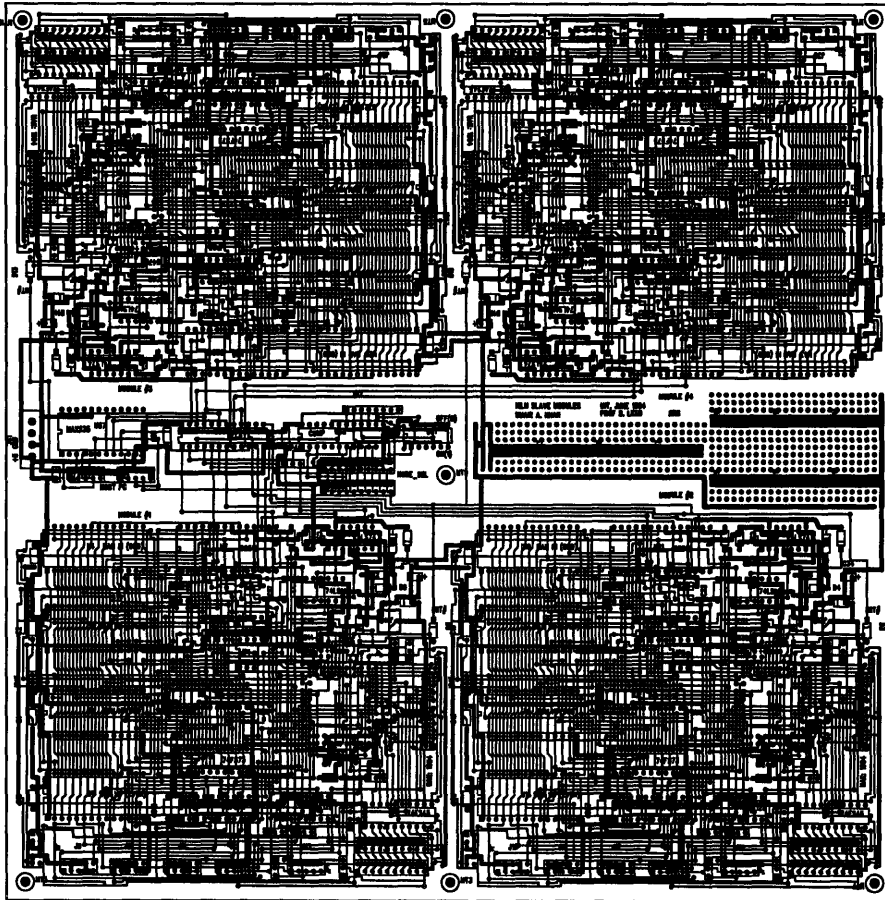


Figure B.7: Slave Board: PCB Component and Solder Sides

Appendix C

PAL Code

This appendix includes the code for all the PALs used for control in the MLM. Pinouts are listed within the files. The functional details of the PALs are discussed in Chapters 3 and 4. We begin with the PALs in the master board.

C.1 Master Board PALs

All microcontrol on the master board is the responsibility of FSMs implemented on PALs - no microprocessor is present on-board. The following PALs are used and their source code is given below, in the order of the listing.

- PAL_AD
- PAL_TR1
- PAL_TR2
- CLK_GEN
- PAL_PC1
- PAL_PC2

Name pal_ad2;
Partno NA;
Date 7/2/94;
Revision 2;
Designer Umair Khan;
Company LEES;
Assembly NILM Front End (Analog Interface);
Location Writes A/D conversion result into SRAM;

```

/*****/
/* */
/* */
/* */
/*****/
/* Allowable Target Device Types: GAL22V10 */
/*****/

```

/ Inputs **/**

```

Pin 1 = clk ; /* */
Pin 2 = lint ; /* */
Pin 3 = pal_tr_busy ; /* */
Pin 4 = adcnt_lsb ; /* */
Pin 5 = adcnt_2ndlsb ; /* */
Pin 6 = adcnt_3rdlsb ; /* */
Pin 7 = lint2 ; /* */
Pin 13 = !reset ; /* */

```

/ Outputs **/**

```

Pin [14..17] = [s3..0] ; /* */
Pin 18 = pal_ad_busy ; /* */
Pin 19 = !clk_ad_cnt ; /* */
Pin 20 = !en_245ad_hi ; /* */
Pin 21 = !en_245ad_lo ; /* */
Pin 22 = !we ; /* */
Pin 23 = !ad_cs_rd ; /* */

```

*/** Declarations and Intermediate Variable Definitions **/*

40

field state = [s3..0];

*/** Logic Equations **/*

sequenced state {

present 'd'0

if int next 'd'15 out pal_ad_busy;

50

default next 'd'0;

present 'd'1

if pal_tr_busy next 'd'1 out pal_ad_busy;

default next 'd'2 out pal_ad_busy out ad_cs_rd out en_245ad_lo;

present 'd'2

next 'd'3 out we out pal_ad_busy out ad_cs_rd out en_245ad_lo;

present 'd'3

next 'd'4 out we out pal_ad_busy out ad_cs_rd out en_245ad_lo;

present 'd'4

60

next 'd'5 out pal_ad_busy out ad_cs_rd out en_245ad_lo;

present 'd'5

next 'd'6 out pal_ad_busy out clk_ad_cnt out ad_cs_rd out en_245ad_lo;

present 'd'6

next 'd'7 out pal_ad_busy out ad_cs_rd;

present 'd'7

next 'd'8 out pal_ad_busy out ad_cs_rd out en_245ad_hi;

present 'd'8

next 'd'9 out we out pal_ad_busy out ad_cs_rd out en_245ad_hi;

present 'd'9

70

next 'd'10 out we out pal_ad_busy out ad_cs_rd out en_245ad_hi;

present 'd'10

next 'd'11 out pal_ad_busy out ad_cs_rd out en_245ad_hi;

present 'd'11

next 'd'12 out pal_ad_busy out clk_ad_cnt out ad_cs_rd out en_245ad_hi;

present 'd'12

next 'd'13 out pal_ad_busy;

```
present 'd'13
    if !adcnt_lsb & !adcnt_2ndlsb & !adcnt_3rdlsb next 'd'0;
    default next 'd'2 out pal_ad_busy out ad_cs_rd out en_245ad_lo;
present 'd'14
    next 'd'0;
present 'd'15
    if int2 next 'd'1 out pal_ad_busy;
    default next 'd'0;
}
```

```

Name    pal_tr1;
Partno  NA;
Date    7/12/93;
Revision 2;
Designer Umair Khan;
Company  LEES;
Assembly NILM Front End (Analog Interface);
Location Transfers SRAM data to Slave 80C196's;

```

```

/*****/
/* */
/* */
/* */
/*****/
/* Allowable Target Device Types: GAL22V10 */
/*****/

```

```

/** Inputs **/

```

```

Pin 1    = clk      ; /* 20
Pin 2    = tr_msb   ; /*
Pin 3    = adcnt_msb ; /*
Pin 4    = pal_ad_busy ; /*
Pin 5    = old_msb  ; /*
Pin 6    = init     ; /*
Pin 13   = !reset   ; /*

```

```

/** Outputs **/

```

```

Pin [14..19] = [s5..0] ; /* 30
Pin 20      = ld_old_ff ;
Pin 23      = !buff_enab ;

```

```

/** Declarations and Intermediate Variable Definitions **/

```

```
field state = [s5..0];
```

40

```
/** Logic Equations **/
```

```
buff_enab.d = state : ['d'3..'d'31];
```

```
sequenced state {
```

```
    present 'd'0
```

```
        if init next 'd'0;
```

50

```
        if adcnt_msb next 'd'1;
```

```
        default next 'd'0 ;
```

```
    present 'd'1
```

```
        if !adcnt_msb next 'd'2;
```

```
        default next 'd'1 ;
```

```
    present 'd'2
```

```
        if pal_ad_busy next 'd'2;
```

```
        if init next 'd'0;
```

```
        default next 'd'3;
```

```
    present 'd'3
```

60

```
        next 'd'4;
```

```
    present 'd'4
```

```
        next 'd'5;
```

```
    present 'd'5
```

```
        next 'd'6;
```

```
    present 'd'6
```

```
        next 'd'7;
```

```
    present 'd'7
```

```
        next 'd'8;
```

```
    present 'd'8
```

70

```
        next 'd'9;
```

```
    present 'd'9
```

```
        next 'd'10;
```

```
    present 'd'10
```

```
        next 'd'11;
```

```
    present 'd'11
```

```
        next 'd'12;
```

```

present 'd'12
    next 'd'13;
present 'd'13
    next 'd'14;
present 'd'14
    next 'd'15;
present 'd'15
    next 'd'16;
present 'd'16
    next 'd'17;
present 'd'17
    next 'd'18;
present 'd'18
    next 'd'19;
present 'd'19
    next 'd'20;
present 'd'20
    next 'd'21;
present 'd'21
    next 'd'22 out ld_old_ff;
present 'd'22
    next 'd'23;
present 'd'23
    next 'd'24;
present 'd'24
    next 'd'25;
present 'd'25
    next 'd'26;
present 'd'26
    next 'd'30;
present 'd'30
    if !tr_msb & old_msb next 'd'0;
    default next 'd'31;
present 'd'31
    if pal_ad_busy next 'd'32;
    default next 'd'12;
present 'd'32
    if pal_ad_busy next 'd'32;
    default next 'd'12;

```

```
present 'd'27
    next 'd'0;
present 'd'28
    next 'd'0;
present 'd'29
    next 'd'0;

present 'd'33
    next 'd'0;
present 'd'34
    next 'd'0;
present 'd'35
    next 'd'0;
present 'd'36
    next 'd'0;
present 'd'37
    next 'd'0;
present 'd'38
    next 'd'0;
present 'd'39
    next 'd'0;
present 'd'40
    next 'd'0;
present 'd'41
    next 'd'0;
present 'd'42
    next 'd'0;
present 'd'43
    next 'd'0;
present 'd'44
    next 'd'0;
present 'd'45
    next 'd'0;
present 'd'46
    next 'd'0;
present 'd'47
    next 'd'0;
```

```
present 'd'48
    next 'd'0;
present 'd'49
    next 'd'0;
present 'd'50
    next 'd'0;
present 'd'51
    next 'd'0;
present 'd'52
    next 'd'0;
present 'd'53
    next 'd'0;
present 'd'54
    next 'd'0;
present 'd'55
    next 'd'0;
present 'd'56
    next 'd'0;
present 'd'57
    next 'd'0;
present 'd'58
    next 'd'0;
present 'd'59
    next 'd'0;
present 'd'60
    next 'd'0;
present 'd'61
    next 'd'0;
present 'd'62
    next 'd'0;
present 'd'63
    next 'd'0;

}
```

Name pal'tr2;
Partno NA;
Date 7/05/94;
Revision 2;
Designer Umair Khan;
Company LEES;
Assembly NILM Front End (Analog Interface);
Location Transfers SRAM data to Slave 80C196's;

```
/*-----*/ 10  
/* */  
/* */  
/* */  
/*-----*/  
/* Allowable Target Device Types: GAL22V10 */  
/*-----*/
```

```
/** Inputs **/
```

```
Pin 1 = clk ; /* 20  
Pin 2 = s5 ; /*  
Pin 3 = s4 ; /*  
Pin 4 = s3 ; /*  
Pin 5 = s2 ; /*  
Pin 6 = s1 ; /*  
Pin 7 = s0 ; /*  
Pin 8 = init ; /*
```

30

```
/** Outputs **/
```

```
Pin 14 = !en_245_trcnt ; /*  
Pin 15 = !en_245_adcncnt ; /*  
/* Be careful here! This is the way the schematics went out*/  
Pin 16 = pal_tr_busy ; /*  
Pin 17 = !clk_trcnt ; /*
```

```

Pin 18      = !ram_oe      ;      /*
Pin 19      = dav3        ;      /*
Pin 20      = dav2        ;      /*
Pin 21      = dav1        ;      /*
Pin 22      = dav0        ;      /*
Pin 23      = int         ;      /*

```

40

```

/** Declarations and Intermediate Variable Definitions **/

```

50

```

field state = [s5..0];

```

```

/** Logic Equations **/

```

```

en_245_adcnt.d = state:['d'0..2] # state:'d'32;
en_245_trcnt.d = state:['d'3..'d'31];
pal_tr_busy.d = state:['d'3..'d'31];
ram_oe.d = state:['d'12..'d'13] # state:['d'15..'d'16] # state:['d'18..'d'19] # state:['d'21..'d'22];
clk_trcnt.d = state:'d'14 # state:'d'17 # state:'d'20 # state:'d'23;
dav0.d = state:['d'13..'d'17];
dav1.d = state:['d'16..'d'20];
dav2.d = state:['d'19..'d'23];
dav3.d = state:['d'22..'d'26];
int.d = state:['d'3..'d'11];

```

Name clk`gen;
 Partno NA;
 Date 6/31/94;
 Revision 2;
 Designer Umair Khan;
 Company LEES;
 Assembly NILM Front End (Analog Interface);
 Location The Sampling Rate Generator for AD7874;

```

/*****/
/*                                     */
/*                                     */
/*                                     */
/*****/
/* Allowable Target Device Types: GAL22V10 */
/*****/
  
```

/ Inputs **/**

```

Pin 1   = clk       ; /*                                     */
Pin 2   = !load    ; /*                                     */
Pin 3   = !reset   ; /*                                     */
  
```

/ Outputs **/**

```

Pin [14..17] = [s3..0] ; /*                                     */
Pin 18      = a0`8254 ; /*                                     */
Pin 19      = a1`8254 ; /*                                     */
Pin 20      = !wr`8254 ; /*                                     */
Pin 21      = prom`a0 ; /*                                     */
Pin 22      = prom`a1 ; /*                                     */
  
```

/ Declarations and Intermediate Variable Definitions **/**

field state = [s3..0];

```
/** Logic Equations */
```

sequenced state –

```

present 'd'0
    if reset next 'd'13;
    next 'd'1;

present 'd'1
    next 'd'2 out a0_8254 out a1_8254;
present 'd'2
    next 'd'3 out a0_8254 out a1_8254 out wr_8254;
present 'd'3
    next 'd'4 out a0_8254 out a1_8254;
present 'd'4
    next 'd'5 out prom_a0;
present 'd'5
    next 'd'6 out prom_a0;
present 'd'6
    next 'd'7 out prom_a0 out wr_8254;
present 'd'7
    next 'd'8 out prom_a0;
present 'd'8
    next 'd'9 out prom_a1;
present 'd'9
    next 'd'10 out prom_a1;
present 'd'10
    next 'd'11 out prom_a1 out wr_8254;
present 'd'11
    next 'd'12 out prom_a1;
present 'd'12
    if load next 'd'13;
    if reset next 'd'13;
    default next 'd'12;
present 'd'13
    if reset next 'd'13;
    if load next 'd'13;

```

```
    default next 'd'0;  
present 'd'14  
    next 'd'0;  
present 'd'15  
    next 'd'0;  
  
}
```

80

Name pcl1a;
Partno NA;
Date 4/26/95;
Revision 2;
Designer Umair Khan;
Company LEES;
Assembly NILM Front End (Analog Interface);
Location Stores A/D conversion in SRAM and sends upon request to PC;

```

/*****/
/*                                     */
/*                                     */
/*                                     */
/*****/
/* Allowable Target Device Types: GAL22V10          */
/*****/

```

/ Inputs **/**

```

Pin 1     = clk           ;   /*                                     */
Pin 2     = pc_mode       ;   /*                                     */
Pin 3     = raw_fetch     ;   /*                                     */
Pin 4     = !ram_we_in   ;   /*                                     */
Pin 5     = !ad_cnt       ;   /*                                     */
Pin 6     = pal_ad_busy   ;   /*                                     */
Pin 7     = pc_fetch     ;   /* Connected by hardware to pin 15 */

```

/ Outputs **/**

```

Pin 14    = misc_ack     ;   /*                                     */
Pin 15    = ff_fetch     ;   /*                                     */
Pin [16..19] = [s3..0]   ;   /*                                     */
Pin 20    = !ram_we      ;   /*                                     */
Pin 21    = !ram_oe      ;   /*                                     */
Pin 22    = mar_cnt      ;   /*                                     */
Pin 23    = !245_enab    ;   /*                                     */

```

```
/** Declarations and Intermediate Variable Definitions **/
```

40

```
field state = [s3..0];
```

```
busy = state'd'14; /** new **/
```

```
/** Logic Equations **/
```

```
245_enab.d = !pc_mode & !busy; /** new **/
```

```
ff_fetch.d = raw_fetch;
```

```
ram_we = ram_we_in & !pc_mode & !busy; /** new **/
```

50

```
sequenced state {
```

```
    present 'd'0
```

```
        next 'd'1;
```

```
    present 'd'1
```

```
        if (pc_mode & !pal_ad_busy) next 'd'8 out misc_ack; /* new */
```

```
        if ram_we_in next 'd'2;
```

```
        default next 'd'1;
```

60

```
    present 'd'2 /*ram_we is still active ie it is a 2 state pulse now*/
```

```
        next 'd'3 out mar_cnt;
```

```
    present 'd'3
```

```
        next 'd'4 out mar_cnt;
```

```
    present 'd'4
```

```
        next 'd'1;
```

```
    present 'd'5
```

```
        next 'd'0;
```

70

```
    present 'd'6
```

```
        next 'd'0;
```

```
    present 'd'7
```

```
        next 'd'0;
```

```

present 'd'8
    if pc_fetch next 'd'9 out ram_oe out misc_ack;           80
    if !pc_mode next 'd'0;
    default next 'd'8 out misc_ack;

present 'd'9
    if !pc_mode next 'd'0;
    default next 'd'10 out ram_oe out misc_ack;

present 'd'10
    if !pc_mode next 'd'0;
    default next 'd'11 out ram_oe out misc_ack;           90

present 'd'11
    if !pc_mode next 'd'0;
    if !pc_fetch next 'd'12 out misc_ack out mar_cnt;
    default next 'd'11 out misc_ack;

present 'd'12
    if !pc_mode next 'd'0;
    default next 'd'13 out misc_ack out mar_cnt;
                                                                 100

present 'd'13
    if !pc_mode next 'd'14; /** new **/
    default next 'd'8;

present 'd'14
    if !pal_ad_busy next 'd'0;           /** new */
    default next 'd'14;

present 'd'15
    next 'd'0;           110
    }

```

Name pc2a;
 Partno NA;
 Date 3/16/95;
 Revision 1;
 Designer Umair Khan;
 Company LEES;
 Assembly NILM Front End (Analog Interface);
 Location Stores A/D conversion in SRAM and sends upon request to PC;

```

/*****/
/*                                     */
/*                                     */
/*                                     */
/*****/
/* Allowable Target Device Types: GAL22V10          */
/*****/
  
```

*/** Inputs **/*

```

Pin 1   = clk      ; /*                                     */
Pin 2   = mar_cnt  ; /*                                     */
Pin [11..7] = [s_in4..0] ; /*                                     */
  
```

*/** Outputs **/*

```

Pin 14  = ram_add0 ; /*                                     */
Pin 15  = ram_add1 ; /*                                     */
Pin 16  = ram_add2 ; /*                                     */
Pin 17  = !4040_clk ; /*                                     */
Pin 18  = 4040_rst ; /*                                     */
Pin 19  = 574_clk  ; /*                                     */
Pin 20  = !574_enab ; /*                                     */
Pin 21  = mar_out  ; /*                                     */
Pin 22  = dav2pc   ; /*                                     */
Pin 23  = ram_add3 ; /*                                     */
  
```

*/** Declarations and Intermediate Variable Definitions **/*

```

field state = [ram_add3..0];
field state_in = [s_in3..0];
reset = state_in.'d'0;

/** Logic Equations **/

4040_rst.d = state_in.'d'0;
574_clk.d = state_in.'d'9;
574_enab.d = state_in:['d'9..'d'11];
dav2pc.d = state_in:['d'10..'d'11];

sequenced state {

    present 'd'0
        if mar_cnt next 'd'9;
        default next 'd'0;

    present 'd'9
        if mar_cnt next 'd'9;
        default next 'd'1;

    present 'd'1
        if mar_cnt next 'd'10;
        if reset next 'd'0;
        default next 'd'1;

    present 'd'10
        if mar_cnt next 'd'10;
        default next 'd'2;

    present 'd'2
        if mar_cnt next 'd'11;
        if reset next 'd'0;
        default next 'd'2;

    present 'd'11
        if mar_cnt next 'd'11;
        default next 'd'3;

```

```

present 'd'3
    if mar_cnt next 'd'12;
    if reset next 'd'0;
    default next 'd'3;
80

present 'd'12
    if mar_cnt next 'd'12;
    default next 'd'4;

present 'd'4
    if mar_cnt next 'd'13;
    if reset next 'd'0;
    default next 'd'4;
90

present 'd'13
    if mar_cnt next 'd'13;
    default next 'd'5;

present 'd'5
    if mar_cnt next 'd'14;
    if reset next 'd'0;
    default next 'd'5;
100

present 'd'14
    if mar_cnt next 'd'14;
    default next 'd'6;

present 'd'6
    if mar_cnt next 'd'15;
    if reset next 'd'0;
    default next 'd'6;

present 'd'15
    if mar_cnt next 'd'15;
    default next 'd'7;
110

present 'd'7
    if mar_cnt next 'd'8 out 4040_clk;
    if reset next 'd'0;

```

```
default next 'd'7;
```

```
present 'd'8
```

```
if mar_cnt next 'd'8;
```

120

```
default next 'd'0;
```

```
}
```

C.2 Slave Board PALs

The microcontrol in each slave module is implemented by a PAL labelled `BUS_CON` in the schematics in Appendix B. The file containing code for this PAL is `slave3.pld`. In addition, two PALs are used in the *Glue Logic Circuitry* of the slave board. The source code for these three PALs is included here in the order of the following listing.

- `BUS_CON`
- `GLUE_RLY`
- `GLUE_CMP`

Name Slave3;
 Partno NA;
 Date 1/21/94;
 Revision 2;
 Designer Umair Khan;
 Company LEES;
 Assembly NILM Slave Processor;
 Location The Euclidean filter computers;

```

/*****/
/* */
/* */
/* */
/*****/
/* Allowable Target Device Types: GAL22V10 */
/*****/
  
```

10

/ Inputs **/**

```

Pin 1   = clk      ; /* */
Pin 2   = stale    ; /* */
Pin 3   = !hlda    ; /* */
Pin [4..11] = [a8..a15] ; /* */
Pin 13  = !reset   ; /* RESPIN */
Pin 22  = !reset2  ; /* */
  
```

20

/ Outputs **/**

```

Pin 14  = !cs510   ; /* */
Pin 15  = !ce_prom ; /* */
Pin 16  = !ce_ram  ; /* */
Pin 17  = !buswidth ; /* */
Pin 18  = state_bit_0 ; /* */
Pin 19  = state_bit_1 ; /* */
Pin 20  = state_bit_2 ; /* */
Pin 21  = !wait    ; /* */
Pin 23  = map      ; /* */
  
```

30

```
/** Declarations and Intermediate Variable Definitions **/
```

```
FIELD memaddr = [a15..8];
```

```
eprom = (!map & memaddr:[2000..27ff]) # memaddr:[0..ff]
        # memaddr:[1000..1dff];
```

```
uart = memaddr:[1e00..1eff];
```

```
wait_1 = stale & !hlda & (wait_2 # eprom);
```

50

```
wait_2 = stale & !hlda & (wait_3 # uart);
```

```
wait_3 = wait_4;
```

```
wait_4 = wait_5;
```

```
wait_5 = wait_6;
```

```
wait_6 = wait_7;
```

```
wait_7 = 'b'0;
```

```
FIELD state_count = [state_bit_0 ..2];
```

```
$DEFINE async_start    'b'000
```

```
$DEFINE hold_2         'b'001
```

60

```
$DEFINE hold_3         'b'011
```

```
$DEFINE hold_4         'b'111
```

```
$DEFINE hold_5         'b'110
```

```
$DEFINE hold_6         'b'100
```

```
$DEFINE hold_7         'b'101
```

```
$DEFINE remove_hold    'b'010
```

```
/* Wait-State Machine */
```

```
SEQUENCE state_count
```

70

```
{
```

```
PRESENT async_start
```

```
IF wait_1 OUT wait;
```

```
IF wait_1 & !wait_2 NEXT remove_hold;
```

```
IF wait_2 NEXT hold_2;
```

```
DEFAULT NEXT async_start;
```

```

PRESENT hold_2
    OUT wait;
    IF wait_3 NEXT hold_3;
    DEFAULT NEXT remove_hold;
80

PRESENT hold_3
    OUT wait;
    IF wait_4 NEXT hold_4;
    DEFAULT NEXT remove_hold;

PRESENT hold_4
    OUT wait;
    IF wait_5 NEXT hold_5;
    DEFAULT NEXT remove_hold;
90

PRESENT hold_5
    OUT wait;
    IF wait_6 NEXT hold_6;
    DEFAULT NEXT remove_hold;

PRESENT hold_6
    OUT wait;
    IF wait_7 NEXT hold_7;
    DEFAULT NEXT remove_hold;
100

PRESENT hold_7
    OUT wait;
    NEXT remove_hold;

PRESENT remove_hold
    NEXT async_start;
110

}

```

*/** Logic Equations **/*


```
map.d = (memaddr:[1000..1dff] & !stale) # map;
map.ar = reset;
map.sp = 'b'0;
map.oe = 'b'1;
ce_prom = (!map & memaddr:[2000..27ff]) # memaddr:[0..ff]
          # memaddr:[1000..1dff];

ce_ram = (map & memaddr:[2000..27ff]) # memaddr:[2800..5fff];
cs510 = memaddr:[1e00..1eff];
buswidth = cs510;

state_bit_0.AR = reset;
state_bit_0.SP = 'b'0;
state_bit_0.OE = 'b'1;
state_bit_1.AR = reset;
state_bit_1.SP = 'b'0;
state_bit_1.OE = 'b'1;
state_bit_2.AR = reset;
state_bit_2.SP = 'b'0;
state_bit_2.OE = 'b'1;
```

Name gluel;
Partno NA;
Date 8/3/94;
Revision 2;
Designer Umair Khan;
Company LEES;
Assembly NILM Slave Processor;
Location Glue Logic Master PAL;

```

/*****/
/*                                     */
/*                                     */
/*                                     */
/*****/
/* Allowable Target Device Types: GAL22V10          */
/*****/

```

/ Inputs **/**

```

Pin 1   = clk      ; /*                                     */
Pin 2   = !reset   ; /*                                     */
Pin 3   = !board_sel ; /*                                     */
Pin [4..5] = [pc0..1] ; /*                                     */
Pin [6..9] = [m_txd0..3] ; /* FROM MAX TXD LINES */
Pin 10  = pc_txd   ; /* FROM PC TXD LINE   */
Pin 11  = m_dcd0   ;
Pin 13  = m_dcd1   ;
Pin 22  = m_dcd3   ;
Pin 23  = m_dcd2   ;

```

/ Outputs **/**

```

Pin [14..17] = [m_rxd0..3] ;
Pin 18  = pc_rxd   ; /* TO PC RXD LINE     */
Pin 19  = !max_not_shut; /*                       */
Pin 21  = pc_dcd   ; /* TO PC DCD LINE     */

```

*/** Declarations and Intermediate Variable Definitions **/* 40

FIELD node_sel = [pc1..0];

node0 = node_sel : 'b'00;

node1 = node_sel : 'b'01;

node2 = node_sel : 'b'10;

node3 = node_sel : 'b'11;

*/** Logic Equations **/* 50

max_not_shut = board_sel;

m_rxd0 = pc_txd # !node0 # !board_sel; */* need "!board_sel" as the PAL */*

m_rxd1 = pc_txd # !node1 # !board_sel; */* generates signals even when the */*

m_rxd2 = pc_txd # !node2 # !board_sel; */* MAX 235 is shutdown, and we want */*

m_rxd3 = pc_txd # !node3 # !board_sel; */* a high on all max_rxd's. */*

/ Don't need a !board_sel for pc_rxd, as that goes thru the MAX235 which is shut down anyway */* 60

pc_rxd = (m_txd0 & node0) # (m_txd1 & node1) # (m_txd2 & node2) # (m_txd3 & node3);

pc_dcd = (m_dcd0 & node0) # (m_dcd1 & node1) # (m_dcd2 & node2) # (m_dcd3 & node3);

```

Name    glue_comp;
Partno  NA;
Date    8/3/94;
Revision 2;
Designer Umair Khan;
Company  LEES;
Assembly NILM Slave Processor;
Location Glue Logic Pal 1 (for comparison of ID bits.);

```

```

/*****/
/*          */
/*          */
/*          */
/*****/
/* Allowable Target Device Types: GAL22V10          */
/*****/

```

```

/** Inputs **/

```

```

Pin 1    = clk      ; /*          */
Pin 2    = lreset   ; /*          */
Pin [6..11] = [pc0..5] ; /* 6 MSB of the PC control byte */
Pin [13..18] = [id0..5] ; /*          */

```

```

/** Outputs **/

```

```

Pin 19 = board_sel ; /* negatively asserted (see below): = 0, if pc = id */
Pin 20 = misc2pc ;

```

30

```

/** Declarations and Intermediate Variable Definitions **/

```

```

/** Logic Equations **/

```

```

board_sel = (pc0 $ id0) # (pc1 $ id1) # (pc2 $ id2) # (pc3 $ id3) # (pc4 $ id4) # (pc5 $ id5);

```


Appendix D

Software for Slave Processors

Three files are included here as samples of the software developed for the three main operations in transient event detection. In addition, the batch file that is used to compile the 80C196 C source code is also given:

- `p0_e1p.c` : V-section Search via Euclidean Filtering.
- `p8_d1p.c` : Tree-Structured Decomposition.
- `p5_c1p.c` : Result Collation.
- `cc.bat` : Batch file that invokes the compiler.

D.1 Code for V-section Search

```
/* p0_e1p.c */

#pragma model(kc)
#pragma interrupt (nmi_master_int = 31)
#pragma interrupt (hsi_data_avail = 2)

/* Slave Processor Code : Pattern recognition and communication. */
/* Suitable for any slave recognizer placed anywhere on the gotcha chain. */
/* Assumes that usec1 in "load" structure has the usec that occurs first ..*/
/* in a data stream. That is, order of usecs must match order of occurrence..*/
/* of usecs in real data */
10

#include<80C196.h>

#define NODE_ID 0
#define TIME_SC 1
#define DATA_SIZE 512
#define INPUT_SIZE (DATA_SIZE*2)
#define T_DETECT +20 /* for template collection mode */
20

#define HSI1 0x01
#define HSI2 0x04
#define HSI3 0x10
#define HSI4 0x40
#define HSI_INPUTS 0 /* for HSI_MODE register */

register char apple[15];
#pragma locate (apple = 0x30)
volatile char input[2*DATA_SIZE]; /* input from master */
30
volatile short data0[DATA_SIZE]; /* converted input data */
volatile short data1[DATA_SIZE]; /* for consecutive block*/
volatile short data2[DATA_SIZE];
volatile short res[DATA_SIZE]; /* results */
volatile short tdata[256];

volatile short template1[]; /* 8 512 point temps., 4 256 point temps. */
```

```

volatile short template2[];
volatile short template3[];
volatile short template4[];
volatile short template5[];
volatile short template6[];
volatile short template7[];
volatile short template8[];
volatile short template9[];
volatile short template10[];
volatile short template11[];
volatile short template12[];

```

40

```

/** locate arrays in memory */

```

50

```

#pragma locate (res = 0x4A00)
#pragma locate (tdata = 0x4E00)
#pragma locate (input = 0x5000)
#pragma locate (data0 = 0x5400)
#pragma locate (data1 = 0x5800)
#pragma locate (data2 = 0x5C00)

```

```

#pragma locate (template1 = 0x3600)    /* 8 BIG TEMPS., 4 SMALL ONES */

```

60

```

#pragma locate (template2 = 0x3800)
#pragma locate (template3 = 0x3A00)
#pragma locate (template4 = 0x3C00)
#pragma locate (template5 = 0x3E00)
#pragma locate (template6 = 0x4000)
#pragma locate (template7 = 0x4200)
#pragma locate (template8 = 0x4400)
#pragma locate (template9 = 0x4600)
#pragma locate (template10 = 0x4700)
#pragma locate (template11 = 0x4800)
#pragma locate (template12 = 0x4900)

```

70

```

/** structure containing info. on loads to be identified by this processor */

```

```

struct load{
    unsigned char used;
    unsigned char mach_no; /*load id */
    unsigned char prev_hs; /*For First Proc. on load chain, no prev_hs..*/

```



```

unsigned char next_hs; /* ..and Prev_got must be set to one for it */
unsigned char no_vsec; /* no of usections associated with load */
volatile short *vs1_add; /* address of 1st vsec */
unsigned short vs1_size; /* size */ 80
volatile short *vs2_add; /* address of 2nd vsec */
unsigned short vs2_size; /* size */
volatile short *vs3_add; /* address of 3rd vsec */
unsigned short vs3_size; /* size */
volatile short *vs4_add; /* address of 4th vsec */
unsigned short vs4_size; /* size */
volatile short *vs5_add; /* address of 5th vsec */
unsigned short vs5_size; /* size */
unsigned char first_got; /* is this the first processor on chain? */
unsigned char last_got; /* is this the last processor on chain? */ 90
unsigned char prev_got; /* Did prev. proc. send the ident. message */
unsigned short prev_time; /* acq_time for prev_got */
unsigned char vs_hits; /* no. of vsecs. identified so far */
unsigned short vs0_loc; /*offset from begin. of inp block for 1st vs*/
unsigned short acq0_time; /* acq_time for first vsec */
unsigned short vs_loc; /*offset from begin. of inp block for Last vs*/
unsigned short thresh; /* error threshold */
short range_hi; /*range within which first and last vsec should be*/
short range_lo; /*range within which first and last vsec should be*/
} 11, 12, 13; 100

```

```

unsigned char new_acq; /*flag*/ /* = 1 if new data block just received */
unsigned short no_of_acq; /*no of input acquisitions made so far*/
unsigned short acq_time;
unsigned short counter;
volatile short *data_point; /* where processed input data resides */
unsigned char data_pos;
unsigned char mode; /* Mode=0 => Norm. Function. Mode=1 => Temp. Collection */
unsigned short temp_acq; /* acquisition no. frozen in temp mode*/ 110
unsigned short *temp_add; /*mem. add where change-of-mean occured*/

```

```

/* Beginning Test memory locations */

```

```

unsigned char hsi_avail_in;

```

```

unsigned char hsi_avail_2;
unsigned char hsi_avail_out;
unsigned char gotcha_in;
unsigned char gotcha_out;
unsigned char euc_in; 120
unsigned char euc_out;
short sr_in; /*unsigned char sr_in;*/
unsigned char sr_out;
unsigned char tempsp;

```

```

#pragma locate (hsi_avail_in = 0x3510) /* hsi_interrupt routine entered */
#pragma locate (hsi_avail_out = 0x3511) /* hsi_interrupt routine exited */
#pragma locate (gotcha_in = 0x3512) /* next proc. comm. routine entered */
#pragma locate (gotcha_out = 0x3513) /* next proc. comm. routine exited */ 130
#pragma locate (euc_in = 0x3514) /* Euclidean Filtering routine entered */
#pragma locate (euc_out = 0x3515) /* Euclidean Filtering routine exited */
#pragma locate (sr_in = 0x3516) /*Serial Comm to Collater routine entered*/
#pragma locate (sr_out = 0x3518) /*Serial Comm to Collater routine exited*/
#pragma locate (hsi_avail_2 = 0x3520) /* validity check routine entered*/
#pragma locate (tempsp = 0x3522)

```

```

/* End of Test memory locations */

```

```

#pragma locate (new_acq = 0x3500) /* = 1 if new data block just received */ 140
#pragma locate (no_of_acq = 0x3502) /*no of input acquisitions made so far*/
#pragma locate (mode = 0x3504) /* = 1 if in template acquisition mode */
#pragma locate (temp_acq = 0x3506) /* acquisition no. frozen in temp mode*/
#pragma locate (temp_add = 0x3508) /*mem. add where change-of-mean occured*/

```

```

void gotcha_comm(struct load *l); /* communicate with next proc., */
void euclidean(struct load *l); /* manage vsec. search */
void load_init(); /* initialize loads */
unsigned char valid(struct load *ld); /* check if contact is valid */
void t_mode(); /* for template acquisition mode */ 150
unsigned short euc(short *temp, unsigned short T_SIZE, unsigned short LIMIT);
void tell_collate(struct load *l); /* serial comm. with collater */

```

```

void load_init()

```

```

{
    /*load l1 */ /*Rapid */
    l1.used = 1;
    l1.mach_no = 1; /*This proc is first proc in load chain of l1 */
    l1.prev_hs= 5; /*initialize prev to garbage val. for First Proc */
    l1.next_hs = 2;
    l1.no_vsec = 2;
    l1.vs1_add = template1; /* address of 1st vsec */
    l1.vs1_size = 16; /* size */
    l1.vs2_add = template2; /* address of 2nd vsec */
    l1.vs2_size = 31; /* size */
    l1.vs3_add = template3; /* address of 3rd vsec */
    l1.vs3_size = 5; /* size */
    l1.vs4_add = template4; /* address of 4th vsec */
    l1.vs4_size = 5; /* size */
    l1.vs5_add = template5; /* address of 5th vsec */
    l1.vs5_size = 5; /* size */
    l1.first_got = 1; /* first processor on load chain*/
    l1.prev_got = l1.first_got;
    l1.last_got = 0;
    l1.vs_hits = 0;
    l1.thresh = 215; /* error threshold */
    l1.range_lo = 0; /*range within which first and last vsec should be*/
    l1.range_hi = 400; /*range within which first and last vsec should be*/

    /*load l2 */ /* computer */
    l2.used = 1;
    l2.mach_no = 3;
    l2.prev_hs= 5; /*initialize prev to garbage val. for First Proc */
    l2.next_hs= 3;
    l2.no_vsec = 1;
    l2.vs1_add = template6; /* address of 1st vsec */
    l2.vs1_size = 22; /* size */
    l2.vs2_add = template7; /* address of 2nd vsec */
    l2.vs2_size = 10; /* size */
    l2.first_got = 1; /* first processor on load chain*/
    l2.prev_got = l2.first_got;
    l2.last_got = 0;
    l2.vs_hits = 0;

```

```

l2.thresh = 200; /* error threshold */
l2.range_lo = 0; /*range within which first and last vsec should be*/
l2.range_hi = 200; /*range within which first and last vsec should be*/

/*load l3 :Instant */ /* trained */
l3.used = 1;
l3.mach_no = 4;
l3.prev_hs= 5; /*initialize prev to garbage val. for First Proc */
l3.next_hs= 4;
l3.no_vsec = 2;
l3.vs1_add = template8; /* address of 1st vsec */
l3.vs1_size = 11; /* size */
l3.vs2_add = template9; /* address of 2nd vsec */
l3.vs2_size = 10; /* size */
l3.first_got = 1; /* first processor on load chain*/
l3.prev_got = l3.first_got;
l3.last_got = 0;
l3.vs_hits = 0;
l3.thresh = 160; /* error threshold */
l3.range_lo = 0; /*range within which first and last vsec should be*/
l3.range_hi = 34; /*range within which first and last vsec should be*/

}

/** NMI ISR */
void nmi_master_int(void)
{
label1:
    while ((ioport2 & 0x08) != 0x08); /* wait for DAV = p2.3 */
    input[counter] = (ioport1 & 0x3f); /***** 6 bits ****/
    while ((ioport2 & 0x08) == 0x08); /* wait for /DAV */
    counter++;
    if (counter < INPUT_SIZE)
        goto label1; /* repeat until 512 points received */

    new_acq = 1;
    no_of_acq++;
    counter = 0;
}

```

```

/** HSI DATA AVAILABLE ISR */
void hsi_data_avail(void)
{
/* This routine responds to hsi events only if a load's "prev_hs" matches */
/* the hs input on which the event takes place */
    unsigned int temptime;
    unsigned char temp, t_ios0;

    /* Test */
    hsi_avail_in++;
    /* End Test */

    temp = hsi_status & 0x55;          /*read hsi line */
    if((temp & 0x01) == 0x01)          /* if hs line 1 */
        { /*output on corresponding hso line*/
            if ((l1.prev_hs == 1)&&(l1.used == 1))
                { /*prev. slave on load chain has sent message*/
                    l1.prev_time = no_of_acq;
                    l1.prev_got = 1;
                }
            else if ((l2.prev_hs == 1)&&(l2.used == 1))
                { /*prev. slave on load chain has sent message*/
                    l2.prev_time = no_of_acq;
                    l2.prev_got =1;
                }
            else if ((l3.prev_hs == 1)&&(l3.used == 1))
                { /*prev. slave on load chain has sent message*/
                    l3.prev_time = no_of_acq;
                    l3.prev_got =1;
                }
            else goto temp2;
            t_ios0 = ios0;          /* handshaking protocol*/
            wsr = 0x0f;
            ios0 = t_ios0 | 0x01;
            wsr = 0;
            while ((hsi_status & 0x02) == 0x02);

```

```

t_ios0 = ios0;
wsr = 0x0f;
ios0 = t_ios0 & 0xfe; /* handshaking protocol completed*/
wsr = 0;
}

temp2:

if((temp & 0x04) == 0x04) /* if hs line 2 */
{
/*output on corresponding hso line*/
if ((l1.prev_hs == 2)&&(l1.used == 1))
{ /*prev. slave on load chain has sent message*/
l1.prev_time = no_of_acq;
l1.prev_got = 1;
}
else if ((l2.prev_hs == 2)&&(l2.used == 1))
{ /*prev. slave on load chain has sent message*/
l2.prev_time = no_of_acq;
l2.prev_got = 1;
}
else if ((l3.prev_hs == 2)&&(l3.used == 1))
{ /*prev. slave on load chain has sent message*/
l3.prev_time = no_of_acq;
l3.prev_got = 1;
}
else goto temp3;
t_ios0 = ios0; /* handshaking protocol*/
wsr = 0x0f;
ios0 = t_ios0 | 0x02;
wsr = 0;
while ((hsi_status & 0x08) == 0x08);
t_ios0 = ios0;
wsr = 0x0f;
ios0 = t_ios0 & 0xfd; /* handshaking protocol completed*/
wsr = 0;
}

```

```

temp3:

```

```

if((temp & 0x10) == 0x10) /* if hs line 3 */
{
    /*output on corresponding hso line*/
    if ((l1.prev_hs == 3)&&(l1.used == 1))
        { /*prev. slave on load chain has sent message*/
            l1.prev_time = no_of_acq;
            l1.prev_got = 1;
        }
    else if ((l2.prev_hs == 3)&&(l2.used == 1))
        { /*prev. slave on load chain has sent message*/
            l2.prev_time = no_of_acq;
            l2.prev_got = 1;
        }
    else if ((l3.prev_hs == 3)&&(l3.used == 1))
        { /*prev. slave on load chain has sent message*/
            l3.prev_time = no_of_acq;
            l3.prev_got = 1;
        }
    else goto temp4;
    t_ios0 = ios0; /* handshaking protocol*/
    wsr = 0x0f;
    ios0 = t_ios0 | 0x04;
    wsr = 0;
    while ((hsi_status & 0x20) == 0x20);
    t_ios0 = ios0;
    wsr = 0x0f;
    ios0 = t_ios0 & 0xfb; /* handshaking protocol completed*/
    wsr = 0;
}

temp4:

if((temp & 0x40) == 0x40) /* if hs line 4 */
{
    /*output on corresponding hso line*/
    if ((l1.prev_hs == 4)&&(l1.used == 1))
        { /*prev. slave on load chain has sent message*/
            l1.prev_time = no_of_acq;
            l1.prev_got = 1;
        }
    else if ((l2.prev_hs == 4)&&(l2.used == 1))

```

```

        { /*prev. slave on load chain has sent message*/
        l2.prev_time = no_of_acq;
        l2.prev_got = 1;
    }
    else if ((l3.prev_hs == 4)&&(l3.used == 1))
        { /*prev. slave on load chain has sent message*/
        l3.prev_time = no_of_acq;
        l3.prev_got = 1;
    }
    else goto temp5;
    t_ios0 = ios0; /* handshaking protocol*/
    wsr = 0x0f;
    ios0 = t_ios0 | 0x08;
    wsr = 0;
    while ((hsi_status & 0x80) == 0x80);
    t_ios0 = ios0;
    wsr = 0x0f;
    ios0 = t_ios0 & 0xf7; /* handshaking protocol completed*/
    wsr = 0;
}

/* Read HSI Time Register */
temp5:
    temptime = hsi_time;

    /* Test */
    hsi_avail_out++;
    /* End Test */
}

main()
{
    unsigned short i, t;
    /* Test Variables*/
    hsi_avail_in=0; hsi_avail_out=0; gotcha_in=0; gotcha_out=0;
    euc_in=0; euc_out=0; sr_in=0; sr_out=0;tempsp=0;temp_acq=0;temp_add=0;
    hsi_avail_2 = 0;
    /* End Test Variable Initialization */
}

```



```

ioport1 = 0xff;
ioport2 = 0x3f; /* bits 6 and 7 are outputs */ 390

counter = 0;
new_acq = 0; /*flag*/
no_of_acq = 0;
acq_time = 0;

/* Port 2 initialization to suit jumper configurations : */
ioport2 = ioport2 | 0x80; /* p2-7 = 1: TXD tristate enable */
ioport2 = ioport2 & 0xbf; /* p2-6 = 0: ioport1 input buff. enable */
ioport2 = ioport2 & 0xdf; /* p2-5 = 0: TS Decomp. interrupt */ 400

wsr = 1;
t2control = 0x01;
wsr = 0;
hsi_mode = HSI_INPUTS; /* hsi input event mode */
ioc0 = 0x55; /* 0b01010101 */
ioc1 = 0x20; /* 0b00100000 */
hso_command = 0x0C; /*reset hso lines */
hso_time = timer1 + 10; 410

int_pending = 0; /* interrupt initialization */
int_mask = 0x84;
baud_rate = 0x67; /* ser_init */
baud_rate = 0x80;
sp_con = 0x01;
/*sbuf = NODE_ID;*/ /* dummy transmission */

load_init(); /* initialize loads */
enable(); /* interrupt initialization */
data_point = data1; /* set input data pointer */ 420
data_pos = 1;

while(1)
{
    if(new_acq) /* if new data, then process */
    {
        if (data_pos == 1)

```

```

        { /*convert two 6 bit points to one 12 bit no*/
            for (i = 0; i < DATA_SIZE; i++)
                {
                    t = input[2*i+1]<<2;
                    if (t > 127)
                        t = t + 0xff00;
                    *(data_point+i) = (t<<4)+input[2*i];
                }
            data_point = data2;
            data_pos = 2;
        }
    else { /*convert two 6 bit points to one 12 bit no*/
        for (i = 0; i < DATA_SIZE; i++)
            {
                t = input[2*i+1]<<2;
                if (t > 127)
                    t = t + 0xff00;
                *(data_point+i) = (t<<4)+input[2*i];
                *(data0 + i) = *(data_point + i);
            }
        data_point = data1;
        data_pos = 1;
    }
    new_acq = 0; /* reset flag */
    if(mode == 1) /* If temp. collection mode... */
        {
            t_mode(); /* ..don't do pattern search */
            continue;
        }
    if (l1.used == 1)
        euclidean(&l1); /*euclidean filtering for l1*/
    if (l2.used == 1)
        euclidean(&l2); /*euclidean filtering for l2*/
    if (l3.used == 1)
        euclidean(&l3); /*euclidean filtering for l3*/
}
if (l1.used == 1)
    {
        if ((l1.vs_hits==l1.no_vsec)&&(l1.prev_got == 1))

```

```

        if(valid(&l1)) /* if load iden. is valid */
            gotcha_comm(&l1);/*send message */
    }
    if (l2.used == 1)
        {
            if ((l2.vs_hits == l2.no_vsec)&&(l2.prev_got == 1))
                if(valid(&l2)) /* if load iden. is valid */
                    gotcha_comm(&l2);/*send message */
        }
    if (l3.used == 1)
        {
            if ((l3.vs_hits==l3.no_vsec)&&(l3.prev_got == 1))
                if(valid(&l3)) /* if load iden. is valid */
                    gotcha_comm(&l3); /*send message */
        }
    }
}

```

*/** Check if Load identification is not false **/*

unsigned char valid(struct load *ld)

```

{
    short temp_diff;
    hsi_avail_2++;
    /* check prev_got */
    if(ld->first_got == 0)
        {
            if(acq_time - ld->prev_time > 1)
                goto bad_end;
        }

    /* see if first to last vsec fall within range and delta */

    if((acq_time - ld->acq0_time) > 1)
        goto bad_end;
    if((acq_time - ld->acq0_time) == 1)
        temp_diff = (ld->vs_loc + 512) - ld->vs0_loc;
    else
        temp_diff = ld->vs_loc - ld->vs0_loc;
}

```

```

        if((temp_diff >= ld->range_lo) && (temp_diff <= ld->range_hi))
            return(1);

bad_end:
    ld->vs_hits = 0;          /* Re-initialize the no. of hits obtained */
    ld->prev_got = ld->first_got; /* Re-initialize the prev_got line */
    return(0);
}

/* This routine does the bookkeeping for usection search and detection */
/* or all loads */
void euclidean(struct load *ld)
{
    unsigned short loc;
    unsigned char i, do_rest;

    do_rest = 0;
    i = ld->vs_hits;
        /* always search for 1st vsec */
        loc = euc(ld->vs1_add, ld->vs1_size, ld->thresh);
        if (loc != 0xffff) /* vsec found! */
            {
                do_rest = 1;
                ld->vs0_loc = loc; /* record for 1st..*/
                ld->acq0_time = no_of_acq; /* vsec */
                ld->vs_hits=1;
            }
        if(ld->no_vsec == ld->vs_hits) /* all vsecs found */
            {
                ld->vs_loc = loc;
                acq_time = no_of_acq;
                return;
            }

    if((i == 1 || do_rest == 1)&&(ld->no_vsec > 1))
        {
            do_rest = 0;
            loc = euc(ld->vs2_add, ld->vs2_size, ld->thresh);
            if (loc != 0xffff) /* vsec found! */

```

```

        {
            do_rest = 1;
            ld->vs_hits++;
        }
    if(ld->no_vsec == ld->vs_hits) /* all vsecs found */
        {
            ld->vs_loc = loc;
            acq_time = no_of_acq;
            return;
        }
}

if((i == 2 || do_rest == 1)&&(ld->no_vsec > 2))
    {
        do_rest = 0;
        loc = euc(ld->vs3_add, ld->vs3_size, ld->thresh);
        if (loc != 0xff) /* vsec found! */
            {
                do_rest = 1;
                ld->vs_hits++;
            }
        if(ld->no_vsec == ld->vs_hits) /* all vsecs found */
            {
                ld->vs_loc = loc;
                acq_time = no_of_acq;
                return;
            }
    }

if((i == 3 || do_rest == 1)&&(ld->no_vsec > 3))
    {
        do_rest = 0;
        loc = euc(ld->vs4_add, ld->vs4_size, ld->thresh);
        if (loc != 0xff) /* vsec found! */
            {
                do_rest = 1;
                ld->vs_hits++;
            }
        if(ld->no_vsec == ld->vs_hits) /* all vsecs found */

```

550

560

570

580

```

        {
        ld->vs_loc = loc;
        acq_time = no_of_acq;
        return;
        }
    }
}

590
if((i == 4 || do_rest == 1)&&(ld->no_vsec > 4))
    {
    loc = euc(ld->vs5_add, ld->vs5_size, ld->thresh);
    if (loc != 0xff)          /* vsec found! */
        {
        do_rest = 1;
        ld->vs_hits++;
        }
    if(ld->no_vsec == ld->vs_hits) /* all vsecs found */
        {
        ld->vs_loc = loc;
        acq_time = no_of_acq;
        return;
        }
    }
}
}

```

```

/* This routine performs the actual Euclidean filtering for each v-section */
unsigned short euc(short *template, unsigned short TEMP_SIZE, unsigned short HI_LIMIT) 610
{
long sum, accum_error;
short dc, *dat;
unsigned short i,j, min, mean_calc, location;

/* Test */
euc_in++;
/* End Test */

sum = 0;
mean_calc = 16384/TEMP_SIZE; /* (2 to the power 14) = 16384 */
location = 0xff;

```

```

if (data_pos == 1)      /* data2 has just been filled */
    dat = data2 - (TEMP_SIZE - 1);
else                    /* else data1 has just been filled */
    dat = data1 - (TEMP_SIZE - 1);

for (j = 0; j < TEMP_SIZE; j++)
    sum+ = *(dat+j);
                                                                    630

for (i = 0; i < DATA_SIZE; i++) /* euclidean filtering the data... */
{
    accum_error = 0;
    dc = (int) ((sum * mean_calc)>>14);
    for (j = 0; j < TEMP_SIZE; j++)
        {
            *(tdata+j) = *(dat+i+j) - dc;
            accum_error+ = abs(*(template+j) - *(tdata+j));
        }
    sum = sum + *(dat+i+j) - *(dat+i);
                                                                    640
    *(res+i) = accum_error;
}

for (i = 0, min = HI_LIMIT; i < DATA_SIZE; i++) /*determine v-sec. location*/
{
    if (*(res+i) < HI_LIMIT)
        {
            if (min > *(res+i))
                {
                    min = *(res+i);
                    location = i;
                                                                    650
                }
        }
}

/* Test */
euc_out++;
/* End Test */

return(location);
                                                                    660
}

```

```

/* Once all vsecs. have been identified and validated, send identification */
/* message to next processor. If this is the last processor, compile a */
/* record and send to collater . gotcha_comm(..) performs these tasks. */
void gotcha_comm(struct load *ld)
{
    int i;
    unsigned char m, t_ios0;
    /* Test */
    gotcha_in++;
    /* End Test */

    ld->vs_hits = 0;          /* Re-initialize the no. of hits obtained */
    ld->prev_got = ld->first_got; /* Re-initialize the prev_got line */
    m = ld->next_hs;
    switch(m)
    {
        case 1:
            t_ios0 = ios0;
            wsr = 0x0f;
            ios0 = t_ios0 | 0x01;
            wsr = 0;
            while ((hsi_status & 0x02) == 0x00);
            if(ld->last_got == 1) /*If last processor on load chain*/
                tell_collate(ld); /*send record to collater */
            t_ios0 = ios0;          /* handshaking protocol*/
            wsr = 0x0f;
            ios0 = t_ios0 & 0xfe;
            wsr = 0;
            while ((hsi_status & 0x02) == 0x02);
            break;          /* handshaking protocol completed*/

        case 2:
            t_ios0 = ios0;
            wsr = 0x0f;
            ios0 = t_ios0 | 0x02;
            wsr = 0;
            while ((hsi_status & 0x08) == 0x00);
            if(ld->last_got == 1) /*If last processor on load chain*/

```



```

        tell_collate(ld); /*send record to collater */
t_ios0 = ios0;          /* handshaking protocol*/
wsr = 0x0f;
ios0 = t_ios0 & 0xfd;
wsr = 0;
while ((hsi_status & 0x08) == 0x08);
break;                /* handshaking protocol completed*/

```

case 3:

```

        t_ios0 = ios0;
wsr = 0x0f;
ios0 = t_ios0 | 0x04;
wsr = 0;
while ((hsi_status & 0x20) == 0x00);
if(ld->last_got == 1) /*If last processor on load chain*/
    tell_collate(ld); /*send record to collater */
t_ios0 = ios0;          /* handshaking protocol*/
wsr = 0x0f;
ios0 = t_ios0 & 0xfb;
wsr = 0;
while ((hsi_status & 0x20) == 0x20);
break;                /* handshaking protocol completed*/

```

case 4:

```

        t_ios0 = ios0;
wsr = 0x0f;
ios0 = t_ios0 | 0x08;
wsr = 0;
while ((hsi_status & 0x80) == 0x00);
if(ld->last_got == 1) /*If last processor on load chain*/
    tell_collate(ld); /*send record to collater */
t_ios0 = ios0;          /* handshaking protocol*/
wsr = 0x0f;
ios0 = t_ios0 & 0xf7;
wsr = 0;
while ((hsi_status & 0x80) == 0x80);
break;                /* handshaking protocol completed*/

```

}

```

    /* Test */
    gotcha_out++;
    /* End Test */
}

```

```

/* Send serial data to collater */
void tell_collate(struct load *ld)

```

```

{
/* A packet consists of: */

```

```

/* 1. processor id      */
/* 2. load id           */
/* 3. location hi byte  */
/* 4. location lo byte  */
/* 5. acq_time hi byte  */
/* 6. acq_time lo byte  */
/* 7. time_scale        */

```

```

char WAIT = 5; /* try 2 */

```

```

/* TEST */

```

```

sr_in++;

```

```

/* END TEST */

```

```

/* enable the LS125 tristate buffer */

```

```

tempsp = sp_stat; /* read sp_stat to clear it */

```

```

ioport2 = ioport2 & 0x7f; /* p2-7 = 0 */

```

```

while (WAIT != 0)

```

```

    WAIT--;

```

```

    sbuf = NODE_ID; /* transmit processor id*/

```

```

    while((sp_stat & 0x20)!=0x20); /* wait to complete trans.*/

```

```

    sbuf = ld->mach_no; /* transmit machine id*/

```

```

    while((sp_stat & 0x20)!=0x20); /* wait to complete trans.*/

```

```

    sbuf = (unsigned char) (ld->vs_loc >> 8); /* transmit loc hi */

```

```

    while((sp_stat & 0x20)!=0x20); /* wait to complete trans.*/

```

```

    sbuf = (unsigned char) ld->vs_loc; /* transmit loc lo*/

```

```

    while((sp_stat & 0x20)!=0x20); /* wait to complete trans. */

```

```

    sbuf = (unsigned char) (acq_time >> 8); /* transmit acq_time hi */

```

```

    while((sp_stat & 0x20)!=0x20); /* wait to complete trans.*/

```

```

sbuf = (unsigned char) acq_time;          /* transmit acq_time lo */
while((sp_stat & 0x20)!=0x20);          /* wait to complete trans. */
sbuf = TIME_SC;                          /* transmit time_sc */
while((sp_stat & 0x20)!=0x20);          /* wait to complete trans. */

/* disable the LS125 tristate buffer */
ioport2 = ioport2 | 0x80; /* p2-7 = 1: TXD tristate enable */
/* TEST */
sr_out++;
/* END TEST */
}
/* This routine handles template collection mode */
void t_mode()
{
    short init, *dat, i, j;
    long int dc;

    if (data_pos == 1) /* data2 has just been filled */
        dat = data2;
    else /* else data1 has just been filled */
        dat = data1;
    init = *(dat+0);
    sr_in = init;
    for(i = 0 ; i < DATA_SIZE-4; i++)
    {
        dc = 0;
        for(j = 0; j < 4; j++)
            dc+= (long int) *(dat+i+j);
        dc = (dc >> 2);
        if((abs(((short) dc) - init)) > T_DETECT)
        {
            temp_acq = no_of_acq;
            temp_add = dat+i;
            while(mode);
            new_acq = 0;
            no_of_acq = 0;
            return;
        }
    }
}

```

}

D.2 Code for Tree-structured Decomposition

```
/* p8_d1p.c */

#pragma model(kc)
#pragma interrupt (nmi_master_int = 31)
#pragma interrupt (hsi_data_avail = 2)

/* Slave Processor Code : Tree Structured Decomposition */
/* This processor may also do pattern recognition */
/* But HSIO0 and HSIO1 not available !!*/

#include<80C196.h>

#define NODE_ID 8
#define TIME_SC 1
#define DATA_SIZE 512
#define DOUT_SIZE (DATA_SIZE/4)
#define T_SIZE 11
#define INPUT_SIZE (DATA_SIZE*2)
#define R_SIZE (DATA_SIZE) /****/
#define HI_LIMIT +200
#define T_DETECT +30 /* for template collection mode */

#define HSI1 0x01
#define HSI2 0x04
#define HSI3 0x10
#define HSI4 0x40
#define HSI_INPUTS (HSI1+HSI2) /* for HSI_MODE register */
#define TIME_COMM 0x02 /* FIGURE OUT time */
#define WAIT_COMM {t_ioc0 = ioc0; ioc0 = t_ioc0 | 0x02; while (timer2 < TIME_COMM);}

register char apple[15];
#pragma locate (apple = 0x30)

volatile char input[INPUT_SIZE]; /* input from master */
volatile short data0[DATA_SIZE]; /* converted input data */
volatile short data1[DATA_SIZE]; /* for consecutive block*/
```

10

20

30

```

volatile short data2[DATA_SIZE];
volatile short result[R_SIZE]; /* results */
volatile short data_out[DOUT_SIZE];
volatile short convtemp[128];
volatile short res[DATA_SIZE]; /* temporary array for decimate() and euc() */
volatile short tdata[256]; /* temporary array for euc() */
volatile short loc[DATA_SIZE];

volatile short template1[256]; /* 4 512 point temps., 5 256 point temps. */
volatile short template2[256];
volatile short template3[256];
volatile short template4[256];
volatile short template5[128];
volatile short template6[128];
volatile short template7[128];
volatile short template8[128];
volatile short template9[128];

#pragma locate (res = 0x4A00)
#pragma locate (tdata = 0x4E00)
#pragma locate (input = 0x5000)
#pragma locate (data0 = 0x5400)
#pragma locate (data1 = 0x5800)
#pragma locate (data2 = 0x5C00)
#pragma locate (convtemp = 0x3600)
#pragma locate (result = 0x3700)
#pragma locate (data_out = 0x3B00)

#pragma locate (template1 = 0x3D00) /* 4 BIG TEMPS., 5 SMALL ONES */
#pragma locate (template2 = 0x3F00)
#pragma locate (template3 = 0x4100)
#pragma locate (template4 = 0x4300)
#pragma locate (template5 = 0x4500)
#pragma locate (template6 = 0x4600)
#pragma locate (template7 = 0x4700)
#pragma locate (template8 = 0x4800)
#pragma locate (template9 = 0x4900)

struct load{

```

```

unsigned char used;
unsigned char mach_no; /*load id */
unsigned char prev_hs; /* For First Proc. on load chain, no prev_hs..*/
unsigned char next_hs; /* ..and Prev_got must be set to one for it */ 80
unsigned char no_vsec; /* no of vsections associated with load */
volatile short *vs1_add; /* address of 1st vsec */
unsigned short vs1_size; /* size */
volatile short *vs2_add; /* address of 2nd vsec */
unsigned short vs2_size; /* size */
volatile short *vs3_add; /* address of 3rd vsec */
unsigned short vs3_size; /* size */
volatile short *vs4_add; /* address of 4th vsec */
unsigned short vs4_size; /* size */
volatile short *vs5_add; /* address of 5th vsec */ 90
unsigned short vs5_size; /* size */
unsigned char first_got; /* is this the first processor on chain? */
unsigned char last_got; /* is this the last processor on chain? */
unsigned char prev_got; /* Did prev. proc. send the ident. message */
unsigned char vs_hits; /* no. of vsecs. identified so far */
unsigned short vs_loc; /*offset from begin. of inp block for Last vs*/
} 11, 12, 13;

```

```

unsigned char new_acq; /*flag*/ /* = 1 if new data block just received */ 100
unsigned short no_of_acq; /*no of input acquisitions made so far*/
unsigned short acq_time;
unsigned short counter;
short *data_point; /* where processed input data resides */
unsigned char data_pos;
unsigned char mode; /* Mode=0 => Norm. Function. Mode=1 => Temp. Collection */
unsigned short temp_acq; /* acquisition no. frozen in temp mode*/
unsigned short *temp_add; /*mem. add where change-of-mean occured*/

```

```

#pragma locate (new_acq = 0x3500) 110
#pragma locate (no_of_acq = 0x3502)
#pragma locate (mode = 0x3504)
#pragma locate (temp_acq = 0x3506)
#pragma locate (temp_add = 0x3508)
/* Beginning Test memory locations */

```

```

unsigned char hsi_avail_in; /* hsi_interrupt routine entered */
unsigned char hsi_avail_out; /* hsi_interrupt routine exited */
unsigned char gotcha_in; /* next proc. comm. routine entered */
unsigned char gotcha_out; /* next proc. comm. routine exited */ 120
unsigned char euc_in; /* Euclidean Filtering routine entered */
unsigned char euc_out; /* Euclidean Filtering routine exited */
unsigned char sr_in; /*Serial Comm to Collater routine entered*/
unsigned char sr_out; /*Serial Comm to Collater routine exited*/
unsigned char temp;
unsigned char master_in; /* Data Comm. to event detector routine entered*/
unsigned char master_out; /* Data Comm. to event detector routine exited*/

#pragma locate (hsi_avail_in = 0x3510)
#pragma locate (hsi_avail_out = 0x3511) 130
#pragma locate (gotcha_in = 0x3512)
#pragma locate (gotcha_out = 0x3513)
#pragma locate (euc_in = 0x3514)
#pragma locate (euc_out = 0x3515)
#pragma locate (sr_in = 0x3516)
#pragma locate (sr_out = 0x3517)
#pragma locate (master_in = 0x3518)
#pragma locate (master_out = 0x3519)
#pragma locate (temp = 0x3520)

140
/* End of Test memory locations */

void init(); /* 80C196 initializations*/
void load_init(); /* initialize loads */
void euclidean(struct load *l); /* manage usec. search */
void t_mode(); /* for template acquisition mode */
unsigned short euc(short *temp, unsigned short t_size);
void gotcha_comm(struct load *l); /* communicate with next proc., */
void tell_collate(struct load *l); /* serial comm. with collater */
void conv(); /* convolution for low-pass filtering */ 150
void s_decimate(); /* simplified decimation */
void decimate(unsigned short *cnt); /* adapt decimate*/
void master(); /* Data Comm. to event detector routine*/

```



```

void load_init()
{
    /*load l1 */
    l1.used = 0; /* load not used */
    l1.mach_no = 1;
    l1.prev_hs= 3;
    l1.next_hs= 4;
    l1.no_vsec = 2;
    l1.vs1_add = template1; /* address of 1st vsec */
    l1.vs1_size = 4; /* size */
    l1.vs2_add = template2; /* address of 2nd vsec */
    l1.vs2_size = 16; /* size */
    l1.last_got = 0;
    l1.vs_hits = 0;

    /*load l2 */
    l2.used = 0; /* load not used */
    l2.mach_no = 4;
    l2.prev_hs= 3;
    l2.next_hs= 4;
    l2.no_vsec = 1;
    l2.vs1_add = template3; /* address of 1st vsec */
    l2.vs1_size = 10; /* size */
    l2.last_got = 1;
    l2.vs_hits = 0;

    /*load l3 : Only two loads can be active at a time due to */
    /* only HSIO2 and 3 being available */
    l3.used = 0; /* load not used */
    l3.mach_no = 4;
    l3.prev_hs= 5;
    l3.next_hs= 5;
    l3.no_vsec = 1;
    l3.vs1_add = template4; /* address of 1st vsec */
    l3.vs1_size = 10; /* size */
    l3.last_got = 1;
    l3.vs_hits = 0;
}

```

```

/* 80C196 initializations*/
void init()
{
    /* Testt Variables */
    hsi_avail_in=0; hsi_avail_out=0; gotcha_in=0; gotcha_out=0;
    euc_in=0; euc_out=0; sr_in=0; sr_out=0; master_in=0; master_out=0;
    /* End Test Variable Initialization */

    counter = 0;
    new_acq = 0;    /*flag*/
    no_of_acq = 0;
    acq_time = 0;

    ioport1 = 0xff;
    ioport2 = 0x3f; /* bits 6 and 7 are outputs */
    /* Port 2 initialization to suit jumper configurations : */
    ioport2 = ioport2 | 0x80; /* p2-7 = 1: TXD tristate enable */
    ioport2 = ioport2 & 0xbf; /* p2-6 = 0: ioport1 input buff. enable */
    /*ioport2 = ioport2 & 0xdf;*/ /* p2-5= 0: TS Decomp. DAV */

    wsr = 1;          /* Timer 2 clock source */
    t2control = 0x01;
    wsr = 0;
    /* ioc2 = 0x40; */ /* timer 2 config */
    hsi_mode = HSI_INPUTS; /* hsi input event mode */
    ioc0 = 0x55; /* 0b01010101 */
    ioc1 = 0x20; /* 0b00100000 */
    hso_command = 0x0C; /*reset hso lines */
    hso_time = timer1 + 10;

    int_pending = 0; /* interrupt initialization */
    int_mask = 0x84; /* 10000100 */
    baud_rate = 0x67; /* ser_init */
    baud_rate = 0x80;
    sp_con = 0x01;
    /*sbuf = NODE_ID;*/ /* dummy transmission */

    load_init(); /* initialize loads */

```

```

enable();          /* interrupt initialization */
data_point = data1; /* set input data pointer */
data_pos = 1;
}

/** NMI ISR */
void nmi_master_int(void)                                240
{
label1:
    while ((ioport2 & 0x08) != 0x08);          /* wait for DAV = p2.3 */
    input[counter] = (ioport1 & 0x3f);          /***** 6 bits *****/
    while ((ioport2 & 0x08) == 0x08); /* wait for /DAV */
    counter++;
    if (counter < INPUT_SIZE)
        goto label1;

    new_acq = 1;                                     250
    no_of_acq++;
    counter = 0;
}

/** HSI DATA AVAILABLE ISR */
void hsi_data_avail(void)
{
/* This routine responds to hsi events only if a load's "prev_hs" matches */
/* the hs input on which the event takes place */
    unsigned int temptime;                            260
    unsigned char temp, t_ios0;

/* Test */
    hsi_avail_in++;
/* End Test */

    temp = hsi_status & 0x55;                          /*read hsi line */
    if((temp & 0x01) == 0x01)                          /* if hs line 1 */
        {
            /*output on corresponding hso line*/
            if ((l1.prev_hs == 1)&&(l1.used == 1))        270
                l1.prev_got = 1;
        }
}

```

```

else if ((l2.prev_hs == 1)&&(l2.used == 1))
    l2.prev_got = 1;
else if ((l3.prev_hs == 1)&&(l3.used == 1))
    l3.prev_got = 1;
else goto temp2;
t_ios0 = ios0; /* handshaking protocol*/
wsr = 0x0f;
ios0 = t_ios0 | 0x01;
wsr = 0;
while ((hsi_status & 0x02) == 0x02);
t_ios0 = ios0;
wsr = 0x0f;
ios0 = t_ios0 & 0xfe; /* handshaking protocol completed*/
wsr = 0;
}

```

temp2:

```

if((temp & 0x04) == 0x04)
{
    /*output on corresponding hso line*/
    if ((l1.prev_hs == 2)&&(l1.used == 1))
        l1.prev_got = 1;
    else if ((l2.prev_hs == 2)&&(l2.used == 1))
        l2.prev_got = 1;
    else if ((l3.prev_hs == 2)&&(l3.used == 1))
        l3.prev_got = 1;
    else goto temp3;
    t_ios0 = ios0; /* handshaking protocol*/
    wsr = 0x0f;
    ios0 = t_ios0 | 0x02;
    wsr = 0;
    while ((hsi_status & 0x08) == 0x08);
    t_ios0 = ios0;
    wsr = 0x0f;
    ios0 = t_ios0 & 0xfd; /* handshaking protocol completed*/
    wsr = 0;
}

```

temp3:

```

if((temp & 0x10) == 0x10)
    {          /*output on corresponding hso line*/
if ((l1.prev_hs == 3)&&(l1.used == 1))
        l1.prev_got = 1;
else if ((l2.prev_hs == 3)&&(l2.used == 1))
        l2.prev_got = 1;
else if ((l3.prev_hs == 3)&&(l3.used == 1))
        l3.prev_got = 1;
else goto temp4;
t_ios0 = ios0; /* handshaking protocol*/
wsr = 0x0f;
ios0 = t_ios0 | 0x04;
wsr = 0;
while ((hsi_status & 0x20) == 0x20);
t_ios0 = ios0;
wsr = 0x0f;
ios0 = t_ios0 & 0xfb; /* handshaking protocol completed*/
wsr = 0;
    }

```

320

330

temp4:

```

if((temp & 0x40) == 0x40)
    {          /*output on corresponding hso line*/
if ((l1.prev_hs == 4)&&(l1.used == 1))
        l1.prev_got = 1;
else if ((l2.prev_hs == 4)&&(l2.used == 1))
        l2.prev_got = 1;
else if ((l3.prev_hs == 4)&&(l3.used == 1))
        l3.prev_got = 1;
else return;
t_ios0 = ios0; /* handshaking protocol*/
wsr = 0x0f;
ios0 = t_ios0 | 0x08;
wsr = 0;
while ((hsi_status & 0x80) == 0x80);
t_ios0 = ios0;
wsr = 0x0f;
    }

```

340

```

        ios0 = t_ios0 & 0xf7; /* handshaking protocol completed*/
        wsr = 0;
    }

/* Read HSI Time Register */
    temptime = hsi_time;

    /* Test */
    hsi_avail_out++;
    /* End Test */
}

main()
{
    unsigned short i, count;
    short t;

    init();

    while(1)
    {
        if(new_acq) /* if new data, then process */
        { /*convert two 6 bit points to one 12 bit no.*/
            if (data_pos == 1)
            {
                for (i = 0; i < DATA_SIZE; i++)
                {
                    t = input[2*i+1]<<2;
                    if (t > 127)
                        t = t + 0xff00;
                    *(data_point+i) = (t<<4)+input[2*i];
                }
                data_point = data2;
                data_pos = 2;
            }
            else { /*convert two 6 bit points to one 12 bit no*/
                for (i = 0; i < DATA_SIZE; i++)
                {

```

```

        t = input[2*i+1]<<2;
        if (t > 127)
            t = t + 0xff00;
        *(data_point+i) = (t<<4)+input[2*i];
        *(data0 + i) = *(data_point +i);
    }
    data_point = data1;
    data_pos = 1;
}
new_acq = 0; /* reset flag */

conv(); /* convolution for low-pass filtering */
s_decimate(); /* simple decimate*/
/*decimate(Bcount);*/ /* adapt decimate*/
master(); /* Data Comm. to event detector routine*/
if(mode == 1) /* If temp. collection mode... */
{
    t_mode(); /* ..don't do pattern search */
    continue; /* ..don't do pattern search */
}
if (l1.used == 1)
    euclidean(&l1); /*euclidean filtering for l1*/
if (l2.used == 1)
    euclidean(&l2); /*euclidean filtering for l2*/
if (l3.used == 1)
    euclidean(&l3); /*euclidean filtering for l3*/
}

if (l1.used == 1)
{
    if ((l1.vs_hits==l1.no_vsec)&&(l1.prev_got == 1))
        gotcha_comm(&l1); /*send message */
}
if (l2.used == 1)
{
    if ((l2.vs_hits == l2.no_vsec)&&(l2.prev_got == 1))
        gotcha_comm(&l2); /*send message */
}
if (l3.used == 1)

```

```

        {
            if ((l3.vs_hits==l3.no_vsec)&&(l3.prev_got == 1))
                gotcha_comm(&l3); /*send message */
        }

    }

}

/* convolution for low-pass filtering */
void conv()
{
    long int sum;
    short i, j, *dat;

    if (data_pos == 1) /* data2 has just been filled */
        dat = data2 - (T_SIZE - 1);
    else /* else data1 has just been filled */
        dat = data1 - (T_SIZE - 1);

    for (i = 0; i < R_SIZE; i++) /* low pass filtering ... */
    {
        sum = 0;
        for (j = 0; j < T_SIZE; j++)
            sum += (((long int)*(dat+i+j))) * ((long int) convtemp[j]);
        result[i] = (short) (sum>>14);
    }
}

/* adapt decimate*/
void decimate(unsigned short *dcount) /* make it take DATASIZE as arg etc.*/
{
    long suma, sumb;
    int nsegs, i, j, num1, count=0;
    short *dat; /* loc[DATA_SIZE] IS res[DATA_SIZE] */

    if (data_pos == 1) /* data2 has just been filled */
        dat = data2 - (T_SIZE - 1);
    else /* else data1 has just been filled */

```

430

440

450

460


```

    dat = data1 - (T_SIZE - 1);

nsegs = 1;
res[nsegs-1] = 0;
num1 = *(dat+0); /* ??????*/
j = 0;

while (j < DATA_SIZE-14)
    {
    while (abs(num1 - result[j]) <= 5)
        j+= 2;
    while (abs(num1 - result[j]) > 5)
        j+= 2;
    if (j < DATA_SIZE-13)
        {
        if (j > res[nsegs-1])
            {
            res[nsegs] = j+15;
            nsegs++;
            }
        }
    }

res[nsegs] = DATA_SIZE - 14;
nsegs++;

for (j =0; j < nsegs - 1; j++)
    {
    suma = 0; sumb = 0;
    for (i = res[j]; i < res[j+1]; i+=2)
        {
        suma += result[i]*result[i];
        sumb += result[i+1] * result[i+1];
        }
    if (suma > sumb)
        {
        for(i = res[j]; i < res[j+1]; i += 2)
            {
            data_out[count] = result[i];

```

```

        count++;
    }
}
else
{
    for(i = res[j]; i < res[j+1]; i += 2)
    {
        data_out[count] = result[i+1];
        count++;
    }
}
}
*dcount = count;
}

/* simple decimate*/
void s_decimate() /* simple decimator: downsampling by 4, even samples */
{
    unsigned int count,i;

    for (i = 0, count = 0 ; count < DOUT_SIZE; count++, i+= 4) /** 2 **/
    data_out[count] = (result[i]>>1);
}

/* Data Comm. to event detector routine*/
void master()
{
    unsigned short i;
    unsigned char t_ioc0, t_ios0;

    master_in++;

    ioport2 = ioport2 | 0x40; /* p2-6 = 1: ioport1 input buff. disable */
    ioport1 = 0;
    t_ios0 = ios0;
    wsr = 15;
    ios0 = t_ios0 | 0x02; /* INT = HSO1 */
    wsr = 0;
    WAIT_COMM

```

510

520

530

540

```

WAIT_COMM
t_ios0 = ios0;
wsr = 15;
ios0 = t_ios0 & 0xfd;    /* INT = HSO1 */
wsr = 0;

                                                                    550

for(i = 0; i < DOUT_SIZE; i++)
{
    WAIT_COMM
    ioport1 = (unsigned char) (*(data_out+i) & 0x3F); /* 6 LSB */
    t_ios0 = ios0;
    wsr = 15;
    ios0 = t_ios0 | 0x01;    /* DAV = HSO0 */
    wsr = 0;
    WAIT_COMM
    t_ios0 = ios0;
    wsr = 15;
    ios0 = t_ios0 & 0xFE;    /* DAV = HSO0 */
    wsr = 0;
    WAIT_COMM
    WAIT_COMM
    ioport1 = (unsigned char) ((*(data_out+i) & 0x0fc0) >> 6); /* 6 MSB */
    t_ios0 = ios0;
    wsr = 15;
    ios0 = t_ios0 | 0x01;    /* DAV = HSO0 */
    wsr = 0;
    WAIT_COMM
    t_ios0 = ios0;
    wsr = 15;
    ios0 = t_ios0 & 0xFE;    /* DAV = HSO0 */
    wsr = 0;
    WAIT_COMM

}

ioport1 = 0xff;
ioport2 = ioport2 & 0xbf; /* p2-6 = 0: ioport1 input buff. enable */
master_out++;
}
                                                                    580

```

```

/* This routine does the bookkeeping for vsection search and detection */
/* or all loads */
void euclidean(struct load *ld)
{
unsigned short loc;
unsigned char i, do_rest;
590

do_rest = 0;
i = ld->vs_hits;
/* always search for 1st vsec */
loc = euc(ld->vs1_add, ld->vs1_size);
if (loc != 0xff) /* vsec found! */
{
do_rest = 1;
ld->vs_hits++;
}
600
if(ld->no_vsec == ld->vs_hits) /* all vsecs found */
{
ld->vs_loc = loc;
acq_time = no_of_acq;
return;
}

if(i == 1 || do_rest == 1)
{
do_rest = 0;
610
loc = euc(ld->vs2_add, ld->vs2_size);
if (loc != 0xff) /* vsec found! */
{
do_rest = 1;
ld->vs_hits++;
}
if(ld->no_vsec == ld->vs_hits) /* all vsecs found */
{
ld->vs_loc = loc;
acq_time = no_of_acq;
620
return;
}
}

```

```

}

if(i == 2 || do_rest == 1)
{
do_rest = 0;
loc = euc(ld->vs3_add, ld->vs3_size);
if (loc != 0xff)          /* vsec found! */
{
do_rest = 1;
ld->vs_hits++;
}
if(ld->no_vsec == ld->vs_hits) /* all vsecs found */
{
ld->vs_loc = loc;
acq_time = no_of_acq;
return;
}
}

if(i == 3 || do_rest == 1)
{
do_rest = 0;
loc = euc(ld->vs4_add, ld->vs4_size);
if (loc != 0xff)          /* vsec found! */
{
do_rest = 1;
ld->vs_hits++;
}
if(ld->no_vsec == ld->vs_hits) /* all vsecs found */
{
ld->vs_loc = loc;
acq_time = no_of_acq;
return;
}
}

if(i == 4 || do_rest == 1)
{
loc = euc(ld->vs5_add, ld->vs5_size);

```

630

640

650

660

```

        if (loc != 0xff)          /* vsec found! */
            {
                do_rest = 1;
                ld->vs_hits++;
            }
        if(ld->no_vsec == ld->vs_hits) /* all vsecs found */
            {
                ld->vs_loc = loc;
                acq_time = no_of_acq;
                return;
            }
    }
}

```

/ This routine performs the actual Euclidean filtering for each v-section */*

```

unsigned short euc(short *template, unsigned short TEMP_SIZE)
{
long sum, accum_error;
short dc, *dat;
unsigned short i,j, min, mean_calc, location;

    /* Test */
    euc_in++;
    /* End Test */

    sum = 0;
    mean_calc = 16384/TEMP_SIZE; /* (2 to the power 14) = 16384 */
    location = 0xff;
    if (data_pos == 1)          /* data2 has just been filled */
        dat = data2 - (TEMP_SIZE - 1);
    else                          /* else data1 has just been filled */
        dat = data1 - (TEMP_SIZE - 1);

    for (j = 0; j < TEMP_SIZE; j++)
        sum+ = *(dat+j);

    for (i = 0; i < DATA_SIZE; i++) /* euclidean filtering the data... */
        {

```

```

    accum_error = 0;
    dc = (int) ((sum * mean_calc)>>14);
    for (j = 0; j < TEMP_SIZE; j++)
    {
        *(tdata+j) = *(dat+i+j) - dc;
        accum_error+ = abs(*(template+j) - *(tdata+j));
    }
    sum = sum + *(dat+i+j) - *(dat+i);
    *(res+i) = accum_error;
}

```

710

```

for (i = 0, min = 250; i < DATA_SIZE; i++) /*determine v-sec. location*/
{
    if (*(res+i) < HI_LIMIT)
    {
        if (min > *(res+i))
        {
            min = *(res+i);
            location = i;
        }
    }
}

```

720

```

}

/* Test */
euc_out++;
/* End Test */

return(location);
}

```

730

```

/* Once all vsecs. have been identified and validated, send identification */
/* message to next processor. If this is the last processor, compile a */
/* record and send to collater . gotcha_comm(..) performs these tasks. */
void gotcha_comm(struct load *ld)
{
    int i;
    unsigned char m, t_ious0;
    /* Test */
    gotcha_in++;
}

```

```
/* End Test */
```

740

```
ld->vs_hits = 0;          /* Re-initialize the no. of hits obtained */
```

```
ld->prev_got = ld->first_got; /* Re-initialize the prev_got line */
```

```
m = ld->next_hs;
```

```
switch(m)
```

```
{
```

```
  case 1:
```

```
    t_ios0 = ios0;
```

```
    wsr = 0x0f;
```

```
    ios0 = t_ios0 | 0x01;
```

750

```
    wsr = 0;
```

```
    while ((hsi_status & 0x02) == 0x00);
```

```
    if(ld->last_got == 1) /*If last processor on load chain*/
```

```
        tell_collate(ld); /*send record to collater */
```

```
    t_ios0 = ios0;          /* handshaking protocol*/
```

```
    wsr = 0x0f;
```

```
    ios0 = t_ios0 & 0xfe;
```

```
    wsr = 0;
```

```
    while ((hsi_status & 0x02) == 0x02);
```

```
    break;          /* handshaking protocol completed*/
```

760

```
  case 2:
```

```
    t_ios0 = ios0;
```

```
    wsr = 0x0f;
```

```
    ios0 = t_ios0 | 0x02;
```

```
    wsr = 0;
```

```
    while ((hsi_status & 0x08) == 0x00);
```

```
    if(ld->last_got == 1) /*If last processor on load chain*/
```

```
        tell_collate(ld); /*send record to collater */
```

```
    t_ios0 = ios0;          /* handshaking protocol*/
```

770

```
    wsr = 0x0f;
```

```
    ios0 = t_ios0 & 0xfd;
```

```
    wsr = 0;
```

```
    while ((hsi_status & 0x08) == 0x08);
```

```
    break;          /* handshaking protocol completed*/
```

```
  case 3:
```

```
    t_ios0 = ios0;
```



```

    wsr = 0x0f;
    ios0 = t_ios0 | 0x04;
    wsr = 0;
    while ((hsi_status & 0x20) == 0x00);
    if(ld->last_got == 1) /*If last processor on load chain*/
        tell_collate(ld); /*send record to collater */
    t_ios0 = ios0;          /* handshaking protocol*/
    wsr = 0x0f;
    ios0 = t_ios0 & 0xfb;
    wsr = 0;
    while ((hsi_status & 0x20) == 0x20);
    break;                  /* handshaking protocol completed*/

case 4:
    t_ios0 = ios0;
    wsr = 0x0f;
    ios0 = t_ios0 | 0x08;
    wsr = 0;
    while ((hsi_status & 0x80) == 0x00);
    if(ld->last_got == 1) /*If last processor on load chain*/
        tell_collate(ld); /*send record to collater */
    t_ios0 = ios0;          /* handshaking protocol*/
    wsr = 0x0f;
    ios0 = t_ios0 & 0xf7;
    wsr = 0;
    while ((hsi_status & 0x80) == 0x80);
    break;                  /* handshaking protocol completed*/
}

/* Test */
gotcha_out++;
/* End Test */

}

/* Send serial data to collater */
void tell_collate(struct load *ld)
{
    /* A packet consists of: */

```

```

/* 1. processor id      */
/* 2. load id          */
/* 3. location hi byte  */
/* 4. location lo byte  */
/* 5. acq_time hi byte  */
/* 6. acq_time lo byte  */
/* 7. time_scale       */

char WAIT = 5; /* try 2 */
/* TEST */
sr_in++;
/* END TEST */
/* enable the LS125 tristate buffer */
tempst = sp_stat; /* read sp_stat to clear it */
ioport2 = ioport2 & 0x7f; /* p2-7 = 0 */

while (WAIT != 0)
    WAIT--;

sbuf = NODE_ID; /* transmit processor id*/
while((sp_stat & 0x20)!=0x20); /* wait to complete trans.*/
sbuf = ld->mach_no; /* transmit machine id*/
while((sp_stat & 0x20)!=0x20); /* wait to complete trans.*/
sbuf = (unsigned char) (ld->vs_loc >> 8); /* transmit loc hi */
while((sp_stat & 0x20)!=0x20); /* wait to complete trans.*/
sbuf = (unsigned char) ld->vs_loc; /* transmit loc lo*/
while((sp_stat & 0x20)!=0x20); /* wait to complete trans. */
sbuf = (unsigned char) (acq_time >> 8); /* transmit acq_time hi */
while((sp_stat & 0x20)!=0x20); /* wait to complete trans.*/
sbuf = (unsigned char) acq_time; /* transmit acq_time lo */
while((sp_stat & 0x20)!=0x20); /* wait to complete trans. */
sbuf = TIME_SC; /* transmit time_sc */
while((sp_stat & 0x20)!=0x20); /* wait to complete trans. */

/* disable the LS125 tristate buffer */
ioport2 = ioport2 | 0x80; /* p2-7 = 1: TXD tristate enable */
/* TEST */
sr_out++;
/* END TEST */

```

```

}

void t_mode()
{
    short init, *dat, i, j;
    long int dc;

    if (data_pos == 1)      /* data2 has just been filled */
        dat = data2;
    else                    /* else data1 has just been filled */
        dat = data1;
    init = *(dat+0);
    sr_in = init;
    for(i = 0 ; i < DATA_SIZE-4; i++)
    {
        dc = 0;
        for(j = 0; j < 4; j++)
            dc+= (long int) *(dat+i+j);
        dc = (dc >> 2);
        if((abs(((short) dc) - init)) > T_DETECT)
        {
            temp_acq = no_of_acq;
            temp_add = dat+i;
            while(mode);
            new_acq = 0;
            no_of_acq = 0;
            return;
        }
    }
}

```

860

870

880

D.3 Code for Result Collation

```
/* p5_c1p.c */

/* For wide collaters, be careful with port 1: */
/* 1. Remove ls245; also disable it via port 2.6 */
/* 2. ONLY DECLARE AS USED THE LOADS THAT ARE ACTUALLY CONNECTED TO PORT1 */
/* OTHERWISE THE COLLATER WILL SEE A ONE (A GOTCHA) ON THE UNUSED, BUT */
/* DECLARED AS USED, PIN */

#pragma model(kc)
#pragma interrupt (nmi_master_int = 31)

/* Slave Processor Code : Collection, collation and pc communication. */
/* Suitable for any slave collater */
/* Assumes that vsec1 in "load" structure has the vsec that occurs first ..*/
/* in a data stream. That is order of vsecs must match order of occurence ..*/
/* of vsecs in reality. */
/* REMEMBER to initialize which of the input lines are being used */

#include<80C196.h>

#define NODE_ID 5
#define DATA_SIZE 512

#define HSI1 0x01
#define HSI2 0x04
#define HSI3 0x10
#define HSI4 0x40
#define HSI_INPUTS 0 /* for HSI_MODE register */

register char apple[15];
#pragma locate (apple = 0x30)

struct load{
    unsigned char new_hit; /* 1 means yes, 0 means no */
    unsigned char used; /* is load used */
    unsigned char got_proc; /* processor which got the final vsecs */
}
```

10

20

30

```

    unsigned char mach_no; /* name of load */
    unsigned char time_scale; /* time scale of identification */
    unsigned char vs_loc_hi; /*offst from begin. of inpt block of Lst vs*/
    unsigned char vs_loc_lo; /*offst from begin. of inpt block of Lst vs*/
    unsigned char acq_time_hi; /* time of event */
    unsigned char acq_time_lo; /* time of event */
} l1, l2, l3, l4, l5, l6, l7, l8;

```

```

/* l1: hsi1, l2: hsi2,...l5: ioport1-1, l6: ioport1-3,...l8: ioport1-7 */

```

```

#pragma locate (l1 = 0x3000)    /* HSIO 0 */
#pragma locate (l2 = 0x3010)    /* HSIO 1 */
#pragma locate (l3 = 0x3020)    /* HSIO 2 */
#pragma locate (l4 = 0x3030)    /* HSIO 3 */
#pragma locate (l5 = 0x3040)    /* P1.0/1 */
#pragma locate (l6 = 0x3050)    /* P1.2/3 */
#pragma locate (l7 = 0x3060)    /* P1.4/5 */
#pragma locate (l8 = 0x3070)    /* P1.6/7 */

```

```

unsigned short no_of_acq; /*no of input acquisitions made so far*/
#pragma locate (no_of_acq = 0x3500) /*no of input acquisitions made so far*/
unsigned char anyhit; /* a load hit received */
#pragma locate (anyhit = 0x3502)

```

```

void ser_comm(struct load *ld, unsigned char c, unsigned char portno);
void init(void);

```

```

unsigned char sr_in; /*Serial Comm to Collater routine entered*/
unsigned char sr_out; /*Serial Comm to Collater routine exited*/
#pragma locate (sr_in = 0x3516)
#pragma locate (sr_out = 0x3517)

```

```

/** NMI ISR **/
void nmi_master_int(void)
{
no_of_acq++;
}

```

```
main()
{
```

```
    unsigned char i, t_ios0;                                80
    /* Test Variables */
    sr_in=0; sr_out=0;
    /* End Test Variable initialization*/
    init();

    while(1)        /* poll all used loads for identification record*/
    {
load1:
        if (l1.used == 0)
            goto load2;                                    90
        if ((hsi_status & 0x02) == 0x02)
            {
                ser_comm(&l1,0x01,0);
                while ((hsi_status & 0x02) == 0x02);
                t_ios0 = ios0;    /* handshaking protocol */
                wsr = 0x0f;
                ios0 = t_ios0 & 0xfe; /* handshaking completed*/
                wsr = 0;
                l1.new_hit =1;
                anyhit = 1; /* to tell PC that a hit was reported */    100
            }

load2:
        if (l2.used == 0)
            goto load3;
        if ((hsi_status & 0x08) == 0x08)
            {
                ser_comm(&l2,0x02,0);
                while ((hsi_status & 0x08) == 0x08);
                t_ios0 = ios0; /* handshaking protocol */    110
                wsr = 0x0f;
                ios0 = t_ios0 & 0xfd; /* handshaking completed*/
                wsr = 0;
                l2.new_hit =1;
                anyhit = 1; /* to tell PC that a hit was reported */
```

```

        }

load3:
    if (l3.used == 0)
        goto load4;
    if ((hsi_status & 0x20) == 0x20)
    {
        ser_comm(&l3,0x04,0);
        while ((hsi_status & 0x20) == 0x20);
        t_ios0 = ios0; /* handshaking protocol */
        wsr = 0x0f;
        ios0 = t_ios0 & 0xfb; /* handshaking completed*/
        wsr = 0;
        l3.new_hit =1;
        anyhit = 1; /* to tell PC that a hit was reported */
    }

load4:
    if (l4.used == 0)
        goto load5;
    if ((hsi_status & 0x80) == 0x80)
    {
        ser_comm(&l4,0x08,0);
        while ((hsi_status & 0x80) == 0x80);
        t_ios0 = ios0; /* handshaking protocol */
        wsr = 0x0f;
        ios0 = t_ios0 & 0xf7; /* handshaking completed*/
        wsr = 0;
        l4.new_hit =1;
        anyhit = 1; /* to tell PC that a hit was reported */
    }

load5:
    if (l5.used == 0)
        goto load6;
    if ((ioport1 & 0x01) == 0x01)
    {
        ser_comm(&l5,0x02,1);
        while ((ioport1 & 0x01) == 0x01);

```

```

        ioport1 = ioport1 & 0xfd; /* handshaking completed*/
        l5.new_hit =1;
        anyhit = 1; /* to tell PC that a hit was reported */
    }

load6:
                                                160
    if (l6.used == 0)
        goto load7;
    if ((ioport1 & 0x04) == 0x04)
        {
            ser_comm(&l6,0x08,1);
            while ((ioport1 & 0x04) == 0x04);
            ioport1 = ioport1 & 0xf7; /* handshaking completed*/
            l6.new_hit =1;
            anyhit = 1; /* to tell PC that a hit was reported */
        }
                                                170

load7:
    if (l7.used == 0)
        goto load8;
    if ((ioport1 & 0x10) == 0x10)
        {
            ser_comm(&l7,0x20,1);
            while ((ioport1 & 0x10) == 0x10);
            ioport1 = ioport1 & 0xdf; /* handshaking completed*/
            l7.new_hit =1;
            anyhit = 1; /* to tell PC that a hit was reported */
        }
                                                180

load8:
    if (l8.used == 0)
        continue;
    if ((ioport1 & 0x40) == 0x40)
        {
            ser_comm(&l8,0x80,1);
            while ((ioport1 & 0x40) == 0x40);
            ioport1 = ioport1 & 0x7f; /* handshaking completed*/
            l8.new_hit =1;
            anyhit = 1; /* to tell PC that a hit was reported */
        }
                                                190

```



```

    }

}
}

```

```
void init(void)
```

200

```
{
```

```
    unsigned char i;
```

```
    no_of_acq = 0;
```

```
    ioport1 = 0x55;
```

```
    ioport2 = 0x3f; /* bits 6 and 7 are outputs */
```

```
    /* Port 2 initialization to suit jumper configurations : */
```

```
    ioport2 = ioport2 | 0x80; /* p2-7 = 1: TXD tristate enable */
```

```
    /*DISABLE (p2-6 =1) ioport1 input buff. enable */
```

210

```
    ioport2 = ioport2 | 0x40; /* p2-6 = 1: ioport1 input buff. enable */
```

```
    ioport2 = ioport2 & 0xdf; /* p2-5 = 0: TS Decomp. interrupt */
```

```
    wsr = 1;
```

```
    t2control = 0x01;
```

```
    wsr = 0;
```

```
    hsi_mode = HSI_INPUTS; /* hsi input event mode .. does not matter*/
```

```
    ioc0 = 0x55; /* 0b01010101 */
```

```
    ioc1 = 0x20; /* 0b00100000 */
```

```
    hso_command = 0x0C; /*reset hso lines */
```

220

```
    hso_time = timer1 + 10;
```

```
    int_pending = 0; /*initialize interrupts */
```

```
    int_mask = 0x80; /* 10000000 */
```

```
    baud_rate = 0x67; /* ser_init */
```

```
    baud_rate = 0x80;
```

```
    sp_con = 0x09;
```

```
    i = sbuf; /* dummy RECEPTION */
```

```
    enable();
```

```
    i = sbuf; /* dummy RECEPTION */
```

230

```
    anyhit = 0; /*initially no hits */
```

```
    ll.new_hit = 0; /*initially no hits */
```

```

12.new_hit = 0;          /*initially no hits */
13.new_hit = 0;          /*initially no hits */
14.new_hit = 0;          /*initially no hits */
15.new_hit = 0;          /*initially no hits */
16.new_hit = 0;          /*initially no hits */
17.new_hit = 0;          /*initially no hits */
18.new_hit = 0;          /*initially no hits */
11.used = 1;            /** 5 loads used here **/
12.used = 1;
13.used = 1;
14.used = 1;
15.used = 1;            /** 5 loads used here **/
16.used = 0;
17.used = 0;
18.used = 0;

}

```

240

250

```

/* routine which receives hit info from event detectors */
void ser_comm(struct load *ld, unsigned char c, unsigned char portno)
{
  unsigned char temp, t_ios0, i;

```

```

/* A packet consists of: */
/* 1. processor id      */
/* 2. machine id        */
/* 3. location hi byte  */
/* 4. location lo byte  */
/* 5. acq_time hi byte  */
/* 6. acq_time lo byte  */
/* 7. time_scale        */

```

260

```

/* TEST */
sr_in++;
/* END TEST */

```

```

temp = sbuf; /* dummy read */
temp = sp_stat; /*reset sp_stat by reading it */

```

270

```

if(portno) /* continue handshaking protocol with slave.. */
    ioport1 = ioport1 | c; /* via port 1 or..*/
else
    {
    t_ios0 = ios0; /* via hsio line */
    wsr = 0x0f;
    ios0 = t_ios0 | c;
    wsr = 0;
}
while((sp_stat & 0x40)!=0x40); /* wait for complete rec. */
ld->got_proc = sbuf; /* transmitted processor id*/
while((sp_stat & 0x40)!=0x40); /* wait for complete rec. */
ld->mach_no = sbuf; /* transmitted machine id*/
while((sp_stat & 0x40)!=0x40); /* wait for complete rec. */
ld->vs_loc_hi = sbuf; /* transmitted vs location (hi)*/
while((sp_stat & 0x40)!=0x40); /* wait for complete rec. */
ld->vs_loc_lo = sbuf; /* transmitted vs location (lo)*/
while((sp_stat & 0x40)!=0x40); /* wait for complete rec. */
ld->acq_time_hi = sbuf; /* transmitted acq time hi */
while((sp_stat & 0x40)!=0x40); /* wait for complete rec. */
ld->acq_time_lo = sbuf; /* transmitted acq time lo*/
while((sp_stat & 0x40)!=0x40); /* wait for complete rec. */
ld->time_scale = sbuf; /* transmitted time scale*/

for( i = 0; i < 20; i++);
/* TEST */
sr_out++;
/* END TEST */
}

```

280

290

300

D.4 Batch File for Code Compilation

The following batch file invokes the compiler (IC-96), the relocater and linker utility (RL-96), and the output-to-hex converter (OH). Carefully note the directory where it expects to find the source file and the directories where the final files are placed.

```
@c:\ic96\bin\ic96 %1.c debug
@c:\ic96\bin\rl96 c:\ic96\lib\cstart.obj, %1.obj, c:\ic96\lib\c96.lib to c:\ecm\out\%1.out stacksize(+10) purge
@c:\ic96\bin\oh c:\ecm\out\%1.out to c:\ecm\hex\%1.hex
@c:\ic96\bin\oh c:\ecm\out\%1.out to c:\umair\animal\%1.hex
```

Appendix E

Software for the Host PC Interface

We include in this appendix the software developed to implement ECM, the PC Interface. In addition, the assembly code added (and other changes made) to the 80C196KC Evaluation Board's RISM to obtain the MLM RISM, are discussed in Section E.2.

E.1 ECM Code

```
/* ECM4.c : Host PC Interface for the ANIMAL */
```

```
/* Version 4.4d */
```

```
/* Date: Apr. 24, 1995 */
```

```
/* ECM2 Additions: */
```

```
/* Download collater code to all 16 processors automatically */
```

```
/* update hits in a history file */
```

```
/* ECM3 Additions */
```

```
/* automatic template loading */
```

10

```
/* Additions in ECM4 */
```

```
/* automate load name collection and collater selection in collater mode */
```

```
/*#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <bios.h>
```

```

#include <conio.h>
#include <graph.h> */

#include "2globcon.h" /* for graphical display */
#include "2globvar.h" /* for graphical display */

/* RISM COMMANDS */
#define SET_DLE_FLAG 0x00
#define TRANSMIT 0x02
#define READ_BYTE 0x04
#define READ_WORD 0x05
#define WRITE_BYTE 0x07
#define WRITE_WORD 0x08
#define LOAD_ADDRESS 0x0A
#define READ_PSW 0x0C
#define WRITE_PSW 0x0D
#define READ_SP 0x0E
#define WRITE_SP 0x0F
#define READ_PC 0x10
#define WRITE_PC 0x11
#define GO 0x12
#define HALT 0x13
#define REPORT_STAT 0x14
#define RESET 0x15

/* For JDR I/O Card Access */
#define MAX_RANGE 0x300
#define IOBASE_ADDR 0x300
#define SLAVE_NODE (IOBASE_ADDR + 0x04)
#define REGA_8255 (IOBASE_ADDR + 0x00)
#define REGB_8255 (IOBASE_ADDR + 0x01)
#define REGC_8255 (IOBASE_ADDR + 0x02)
#define CONT_8255 (IOBASE_ADDR + 0x03)
#define MASTER_SET (IOBASE_ADDR + 0x08)
#define COM1_CONT 0x3FC

#define MODE_8255 0x93
#define SETFETCH {outp(CONT_8255,0x0B);} /* CONTROL BYTE TO SET FETCH:PC5 */
#define CLRFETCH {outp(CONT_8255,0x0A);} /* CONTROL BYTE TO RESET FETCH:PC5 */

```

```

#define SETMODE {outp(CONT_8255,0x09);} /* CONTROL BYTE TO SET MODE:PC4 */
#define CLRMODE {outp(CONT_8255,0x08);} /* CONTROL BYTE TO RESET MODE:PC4 */
#define MISC_MASK 0x02 /* misc is input pc1 */
#define DAV_MASK 0x01 /* dav is input pc0*/
60

#define LOADS 0x08
#define NUM_SCALE 0x02
#define LOAD_NUM (LOADS * NUM_SCALE) /*LOAD_NUM=LOADS*TIME_SCALES*/
#define COLL_NUM 0x04
#define HI_ADDR 0x30 /* high addr byte for packets in collaters */
#define HI_MODE_LOC 0x35 /* High Addr. for "mode" in slaves */
#define LO_MODE_LOC 0x04 /* Low Addr. for "mode" in slaves */

#define ADD_HI_HIT 0x35 /* high addr for hit byte in collaters */
#define ADD_LO_HIT 0x02 /* low addr for hit byte in collaters */
70

#define CHK_WAIT 200000 /* wait in CHECK_STATUS() */
#define NUM_PROCS 16

void normal(); /* for main menu selection 1*/
void echo(); /* for main menu selection 2*/
void glue_io(); /* for main menu selection 3*/
void master_io(); /* for main menu selection 4*/
void master_set(); /* for main menu selection 5*/
void slave_set(); /* for main menu selection 6*/
void rst_slave(); /* for main menu selection 6.3*/
void reset_it(); /* subroutine performing reset*/
void coll_comm(); /* for main menu selection 7*/
void init_proc(); /* for main menu selection 9*/
80

void loadfile(); /* for main menu selection 1.1*/
void load_it(FILE *infile); /* subroutine performing reset*/
void temp(); /* for main menu selection 1.2*/
void loadtemp(); /* for main menu selection 1.2.1*/
90
void temp_it(unsigned char add_hi, unsigned char add_lo, FILE *infile);
void read_range(); /* for main menu selection 1.5*/
void poke_mem(); /* for main menu selection 1.6*/
void make_temp(); /* for main menu selection 1.2.2*/

```

```

/* support routines for rism comm. */
void load_data(unsigned char lo_byte, unsigned char hi_byte);
void load_addr(unsigned char lo_byte, unsigned char hi_byte);
void load_pc(unsigned char lo_byte, unsigned char hi_byte);
void load_psw(unsigned char lo_byte, unsigned char hi_byte);
unsigned char gethex(FILE *infile);
void CHECK_STATUS(); /* check status of com1 */
void REC_WAIT(); /* wait for reception of data from MLM slave */
void DELAY(); /* delay between successive polling of collaters etc.*/
void DELAY2(); /*used in coll_comm() to wait until all hits are received */
int run_stat(); /* check status to see if program running */
unsigned int read_sfr(); /* read SFR : SP, PC, PSW */

/* support routines for retrieving hit info and master board data: */

/* subroutine for retrieving hit info data: */
void get_hit(unsigned char lo_addr, int collater);

void get_byte();
void update(); /* update contacts in contacts.txt */
void history(); /*update history.txt with latest hit */
void armed(); /* collater communication mode */
void wait_exit();
void get_16sec(unsigned int bar); /* retrieve last several seconds of..*/
/* ..acquired data from master */

void mark(); /* external function: */

/* collater communication global variable */
unsigned com1_stat;
unsigned long chk_wait; /*
unsigned char cur_proc; /* currently selected processor */
unsigned char num_coll; /* number of collaters */
unsigned char id[COLL_NUM]; /* collater+load array
unsigned char msmode; /*master mode */

/* A packet, sent on a load hit, consists of: */

/* 0. hit? */

```



```

/* 1. machine id      */
/* 2. time_scale     */
/* 3. location hi byte */
/* 4. location lo byte */
/* 5. acq_time hi byte */
/* 6. acq_time lo byte */
/* IN ADDITION, THE COMPILED RECORD CONSISTS OF:*/
/* 7. current time   */

```

140

```

struct load_hit{
    unsigned char ld_hit; /* load found */
    char name[10]; /*load id */
    unsigned char load_id; /*load id */
    unsigned char scale; /* time scale */
    unsigned char hi_loc; /*offset from begin. of inpt block for Lst vs*/
    unsigned char lo_loc; /*offset from begin. of inpt block for Lst vs*/
    unsigned char hi_time; /* acq time of event */
    unsigned char lo_time; /* acq time of event */
    unsigned int cur_time; /* current acq time */
}packet[COLL_NUM][8];

```

150

```

main()
{

```

```

    char mode;

```

160

```

    _bios_serialcom(_COM_INIT,0,(_COM_CHR8 | _COM_NOPARITY | _COM_STOP1 | _COM_9600));

```

```

    outp(CONT_8255, MODE_8255); /* INITIALIZE PC COMM PORTS ON 8255 ETC.*/

```

```

    CLRFETCH;

```

```

    CLRMODE;

```

```

    for ( ; )

```

```

    {

```

```

        _clearscreen(_GCLEARSCREEN);

```

```

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

```

```

        printf("\t\tECM version 4.4 \t April 12, 1995");

```

```

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

```

```

        printf("\n\n*****          MAIN MENU          *****\n\n");

```

```

        printf("\n\n1. Slave Communication\n");

```

```

printf("2. Slave Diagnostic/Echo\n");
printf("3. Slave Processor Selection\n");
printf("4. Data from Master\n");
printf("5. Master Acquisition Mode Setup\n");
printf("6. Slave Mode Selection\n");
printf("7. Collater Communication\n");
printf("8. Display Channel Data\n");
printf("9. Initialize Slave Processors\n");
printf("E. Exit ECM\n\n");
scanf("%c", &mode);

switch (mode)
{

    case '1':
        normal();
        break;

    case '2':
        echo();
        break;

    case '3':
        glue_io();
        break;

    case '4':
        master_io();
        break;

    case '5' :
        master_set();
        break;

    case '6' :

```

180

190

200

210

```

        slave_set();
    break;

    case '7' :
        coll_comm();
    break;

    case '8' :
        mark();
    break;

    case '9' :
        init_proc();
    break;

    case 'e' :
        return 0;
}
}
}
/* menu selection 2 */
void echo()
{
    unsigned int i,j;
    unsigned char ch, ch1;
    _clearscreen(_GCLEARSCREEN);

    /* Echo mode */
    printf("\n\n*****          ECHO  MODE          *****\n\n");
    printf("\n\nThis is the Echo Mode.  Characters entered are echoed on the board.  \n");
    printf("A '\\\'' or '/' returns you to the main menu.  \n\n");

    ch = run_stat();
    if (ch == 1)
    {
        printf("\n\n\t**** Processor %d Is Running ****\n", cur_proc);
        printf("\t**** Press Any Key To Exit ****\n");

```

```

        while (!kbhit());
        ch = getch();
        return;
    }

    printf("\nResetting Processor %d...", cur_proc);
    load_data(1,0);
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0, RESET);           260
    /* wait */
    for (j = 0; j < 600; j++)
        for (i = 0; i < 32000; i++);
    printf("\nProcessor %d Reset", cur_proc);

    ch1 = _bios_serialcom(_COM_RECEIVE,0,0); /* Dummy Read */

    while(1)
    {
        printf("\nInput Character to be displayed by ecm\n");
        printf("\n* ");
        ch = getchar();
        if (ch == '\n')
            ch = getchar();

        if ((ch == '/') || (ch == '\\'))
            break;

        _bios_serialcom(_COM_SEND,0, ch);
        REC_WAIT();                               280
        ch1 = _bios_serialcom(_COM_RECEIVE,0,0);
        printf("Character Sent = %c (%x). Character Received = %x.\n\n", ch, ch, ch1);
    }
}

/* menu selection 1 */
void normal()
{
    unsigned int i,j, m;                           290

```

```

/* Check if user running. If not then remap to user */
i = run_stat();
if (i != 1)
    {
        CHECK_STATUS();
        _bios_serialcom(_COM_SEND,0, '\\');    /*user mapped */
    }

for(;; )
    {
        _clearscreen(_GCLEARSCREEN);
        printf("\n\n\t***** Normal Mode of Execution *****\n\n");

        printf("\n\n1. Load File \n");
        printf("2. Template Management\n");
        printf("3. Go\n");
        printf("4. Halt\n");
        printf("5. Read Range\n");
        printf("6. Poke Memory\n");
        printf("7. Reset Processor %d\n", cur_proc);
        printf("8. Reset Processor %d and Remap to User\n", cur_proc);
        printf("9. Quit\n\n");

        m = getchar();
        if (m == '\n')
            m = getchar();

        i = 0;
        switch(m)
            {
                case '1' :
                    /* Check if user running. If so then cannot load */
                    i = run_stat();
                    if (i == 1)
                        {
                            printf("\n\nProgram Running - Command Ignored.\n");
                            wait_exit();
                            break;
                        }
            }
    }

```

```

}
loadfile();
break;

case '2' :
    temp();
break;

case '3' :
    i = run_stat();
if (i == 1)
{
    printf("\n\nProgram Running - Command Ignored.\n");
    wait_exit();
    break;
}
CHECK_STATUS();
_bios_serialcom(_COM_SEND,0, GO);
break;

case '4' :
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0, HALT);
break;

case '5':
    read_range();
break;

case '6':
    poke_mem();
break;

case '7':
    printf("\n\nResetting Processor %d...", cur_proc);
load_data(1,0);
CHECK_STATUS();
_bios_serialcom(_COM_SEND,0, RESET);
/* wait */

```



```

        return;
    }
    printf("\nLoading File into Target RAM...");
    load_it(inf);
    printf("\n\nProgram Loaded. \n");
    fclose(inf);
    wait_exit();
}

void load_it(FILE *inf)
{
    int i;
    unsigned char temp, count, type, add_hi, add_lo;

    while (1)
    {
        temp = getc(inf);
        if (temp != ':')
        {
            printf("\n\nERROR: File not in HEX Format. Exiting.\n");
            wait_exit();
            return;
        }

        count = gethex(inf);
        add_hi = gethex(inf);
        add_lo = gethex(inf);
        type = gethex(inf);
        if (type == 0x01)
            break;
        load_addr(add_lo, add_hi);

        for (i = 0; i < count; i++)
        {
            temp = gethex(inf);
            CHECK_STATUS();
            _bios_serialcom(_COM_SEND,0, SET_DLE_FLAG);
            CHECK_STATUS();
            _bios_serialcom(_COM_SEND,0, temp);

```



```

        CHECK_STATUS();
        _bios_serialcom(_COM_SEND,0, WRITE_BYTE);
    }
    temp = gethex(inf);    /* checksum */
    temp = getc(inf);     /* newline or carriage return */
}

/* Reset User PC and User PSW */

load_pc(0x80,0x20);
load_psw(0x80,0x02);
}

/* menu selection 9 */
void init_proc()
{
    int i, j;
    FILE *fopen(), *inf, *flist, *tlist;
    char fname[15], tname[15],k;

    _clearscreen(_GCLEARSCREEN);

    printf("\n\n\t***** Processor Initialization *****\n\n");
    printf("\nTo EXIT: Press 'e' and ENTER.\n");
    printf("\nTO INITIALIZE: Press any other key and then hit ENTER. ");
    scanf("%c", &k);
    scanf("%c", &k);
    if (k == 'e')
        return;

    /* make sure procfile.txt is present : */

    if ((flist = fopen("procfile.txt", "r")) == NULL)
    {
        printf("\n\nCannot access file 'procfile.txt'\n");
        fclose(flist);
        wait_exit();
        return;
    }
}

```

```

/* make sure proctemp.txt is present : */
if ((tlist = fopen("proctemp.txt", "r")) == NULL)
    {
        printf("\n\nCannot access file 'proctemp.txt'\n");           490
        fclose(tlist);
        wait_exit();
        return;
    }

reset_it(); /* resetting all slaves */
for (j = 0; j < 500; j++)
    for (i = 0; i < 32000; i++); /* Wait */
CHECK_STATUS();
outp(SLAVE_NODE, 0); /* select it first*/                               500
for (j = 0; j < 500; j++)
    for (i = 0; i < 32000; i++); /* Wait */

printf("\n\n");
for(i = 0; i < NUM_PROCS; i++)
    {
        CHECK_STATUS();
        if(i != 0)
            outp(SLAVE_NODE, i);                                       510
        DELAY();
        DELAY();
        j = run_stat();
        if (j != 1)
            {
                CHECK_STATUS();
                _bios_serialcom(_COM_SEND,0, '\n'); /*user mapped */
                printf("\nSlave %d User-mapped.\n",i);
            }
        printf("Initializing Slave %d:", i);                             520

        fscanf(flist,"%s", fname);
        printf("\nLoading file %s into Slave %d...",fname,i);
        if ((inf = fopen(fname, "r")) == NULL)

```

```

        {
            printf("\n\nCannot access file '%s'\n",fname);
            wait_exit();
            return;
        }
load_it(inf);
fclose(inf);

/* load template */

for(k = 0x36; ; k= k+2)
    {
        fscanf(tlist,"%s", fname);
        if(strcmp(fname,":end") == 0)
            break;
        printf("\nLoading Template %s at 0x%x00 in Slave %d...",fname,k,i);
        if ((inf = fopen(fname, "r")) == NULL)
            {
                printf("\n\nCannot access file '%s'\n", fname);
                wait_exit();
                return;
            }
        temp_it(k, 0x00, inf);
        fclose(inf);
    }
CHECK_STATUS();
_bios_serialcom(_COM_SEND,0, GO);
printf("Done.\n");
}
CHECK_STATUS();
outp(SLAVE_NODE, cur_proc);
printf("\nSlave Modules Initialized\n");
fclose(flist);
fclose(tlist);
wait_exit();
}

/* menu selection 1.2 */
void temp()

```

530

540

550

560

```

{
unsigned char data, i, m;

_clearscreen(_GCLEARSCREEN);
printf("\n\n\t ***** Template Management *****\n\n");

    printf("\n\nSelect Function:\n\n");
    printf("1. Load Template File\n");
    printf("2. Convert Raw Data to Template\n");
    printf("3. Exit \n\n");

        m = getchar();
    if (m == '\n')
        m = getchar();
    switch(m)
    {
        case '1' :
            loadtemp();
            break;

        case '2':
            make_temp();
            break;

        case '3':
            return;
    }
}

/* menu selection 1.2.1 */
void loadtemp()
{
    unsigned int addr;
    char fname[20];
    unsigned char add_hi, add_lo,temp, temp2;
    FILE *fopen(), *inf;

    printf("Enter Template (.tmp) File to be loaded: ");
    scanf("%s", fname);

```

570

580

590

600

```

if ((inf = fopen(fname, "r")) == NULL)
    {
        printf("\n\nCannot access file '%s'\n", fname);
        wait_exit();
        return;
    }
printf("Enter Address (in hex) at which Template is to be loaded: ");
scanf("%x", &addr);
610

add_hi = ((unsigned char) (addr>>8));
add_lo = ((unsigned char) addr);
temp_it(add_hi, add_lo, inf);
printf("\n\nTemplate Loaded.\n");
wait_exit();
fclose(inf);
}

620

void temp_it(unsigned char add_hi, unsigned char add_lo, FILE *inf)
{
    unsigned char temp, temp2;

    load_addr(add_lo, add_hi);

    while(1)
        {
            temp = getc(inf);
            if (temp != ':')
                break;
            temp = gethex(inf);    /* MSB-- sent later */
            temp2 = gethex(inf);  /* LSB-- sent first */
            load_data(temp2,temp);
            CHECK_STATUS();
            _bios_serialcom(_COM_SEND,0, WRITE_WORD);
            temp = getc(inf); /* Carriage Return */
        }

640
}

```

```

/* menu selection 1.2.2 */
void make_temp()
{
    long accum=0;
    int value[256],dc;
    unsigned int i, j;
    char fname[20], f2name[20];
    unsigned char temp, temp2;
    FILE *fopen(), *inf, *outf;

    printf("\n\nEnter Raw Data File Name: ");
    scanf("%s", fname);
    printf("\n\nEnter Output Data File Name: ");
    scanf("%s", f2name);

    if ((inf = fopen(fname, "r")) == NULL)
    {
        printf("\n\nCannot access file '%s'\n", fname);
        wait_exit();
        return;
    }

    for(i=0; ; i++)
    {
        temp = getc(inf);
        if (temp != ':')
            break;

        temp = gethex(inf);      /* MSB-- sent later */
        temp2 = gethex(inf);    /* LSB-- sent first */
        value[i] = 0;
        value[i] = temp;
        value[i] = ((int) ((value[i] << 8) + temp2));
        accum = accum + value[i];
        temp = getc(inf);
    }

    fclose(inf);
    outf = fopen(f2name, "w");
    dc = (int) ((accum * (16384/i)) >> 14);
    for(j = 0; j < i; j++)

```

650

660

670

680

```

        {
        value[j] = value[j] - dc;
        temp = (unsigned char) value[j];
        temp2 = ((unsigned char) (value[j] >> 8));
        fprintf(outf, ":%2.2X%2.2X\n", (int) temp2, (int) temp);
        }
    fprintf(outf, "END\n");
    fprintf(outf, "\n\nStatistics on Template:\n\n");
    fprintf(outf, "Size = %d\n", i);
    fprintf(outf, "Sum = %d\n", accum);
    fprintf(outf, "DC value = %d\n", dc);
    fclose(outf);
    printf("\n\n Template Data stored in '%s'.", f2name);
    wait_exit();
}

```

/ subroutine for reading hex data from .tmp file */*

```

unsigned char gethex(FILE *infile)
{
    unsigned char ch[2], t, num=0;
    fscanf(infile, "%2s", ch);
    for (t = 0; t < 2; t++) /* Convert both ascii digits to one hex num */
    {
        if (ch[t] > 0x40)
            ch[t] = ch[t] - 55;
        else
            ch[t] = ch[t] - 48;
    }

    num = ch[1] + (ch[0] << 4);
    return(num);
}

```

/ menu selection 1.5*/*

/ subroutine for reading data from slave */*

void read_range()

```

{
    unsigned char com2_stat, lo_byte, hi_byte, range[MAX_RANGE], k;
    short int i, j, st_add, end_add, max_index, temp;

```

```

FILE *fopen(), *outf;
char fname[20];

printf("\nStart Address (in Hex): ");
scanf("%x", &st_add);
printf("\nEnd Address (in Hex): ");
scanf("%x", &end_add);
i = (end_add - st_add);
if ((i > MAX_RANGE) || (i < 0))
    {
        printf("\nERROR: Illegal Address Range.");
        wait_exit();
        return;
    }
lo_byte = (unsigned char) st_add;
hi_byte = (unsigned char) (st_add >> 8);
load_addr(lo_byte, hi_byte);

range[0] = _bios_serialcom(_COM_RECEIVE,0,0); /* dummy read */
for(i = st_add, j = 0; i <= end_add; i++, j++)
    {
        get_byte();
        range[j] = _bios_serialcom(_COM_RECEIVE,0,0);
    }

max_index = j;
printf("\nRead Operation Completed. Press Any Key to display data...\n\n");
while (!kbhit());
k = getch();
_clearscreen(_GCLEARSCREEN);
for(i = 0; i < max_index; )
    {
        printf("\n%4x:\t ", (st_add+i));
        for (j = 0; (j < 16) && (i < max_index); j++, i++)
            printf("%2.2x ",range[i]);
    }

printf("\n\nDo you want to Dump Data to a file [n]? ");
scanf("%c", &k);

```



```

scanf("%c", &k);
if (k != 'y')
    return;

printf("\nEnter File Name: ");
scanf("%s", fname);
outf = fopen(fname, "w");
fprintf(outf, "\nADDR:\t\t\t 16 bytes of Data (LSB First)\n");
fprintf(outf, "-----\n");
for(i = 0; i < max_index; )
    {
        fprintf(outf, "\n%4X:\t ", (st_add+i));
        for (j = 0; (j < 16) && (i < max_index); j++, i++)
            fprintf(outf, "%2.2X ", range[i]);
    }
fclose(outf);

printf("\n\n Data stored in file '%s.'", fname);
wait_exit();
}

/* menu selection 1.6*/
void poke_mem()
{
    unsigned int reg, m;

    for(;; )
        {
            _clearscreen(_GCLEARSCREEN);
            printf("\n\n\t***** Read Special Function Registers *****\n");
            printf("\n\n1.  Fetch SP \n");
            printf("2.  Fetch PC \n");
            printf("3.  Fetch PSW \n");
            printf("4.  Exit\n\n");

            m = getchar();
            if (m == '\n')
                m = getchar();

```

760

770

780

790

```

switch(m)
{
    case '1' :
        reg = read_sfr(READ_SP);
        printf("\n\n SP = %x ", reg);
        break;

    case '2' :
        reg = read_sfr(READ_PC);
        printf("\n\n PC = %x ", reg);
        break;

    case '3' :
        reg = read_sfr(READ_PSW);
        printf("\n\n PSW = %x ", reg);
        break;

    case '4' :
        return;
}
wait_exit();
}

```

/ read in Special Function Registers, SFRs, in menu 1.6*/*

```

unsigned int read_sfr(unsigned char com_name)

```

```
{
```

```
    unsigned int sreg = 0;
```

```
    unsigned char lsb_ch;
```

```
    lsb_ch = _bios_serialcom(_COM_RECEIVE,0,0); /* Dummy Read */
```

```
    CHECK_STATUS();
```

```
    _bios_serialcom(_COM_SEND,0, com_name);
```

```
    CHECK_STATUS();
```

```
    _bios_serialcom(_COM_SEND,0, TRANSMIT);
```

```
    REC_WAIT();
```

```
    lsb_ch = _bios_serialcom(_COM_RECEIVE,0,0);
```

```
    CHECK_STATUS();
```

```

    _bios_serialcom(_COM_SEND,0, TRANSMIT);
    REC_WAIT();
    sreg = _bios_serialcom(_COM_RECEIVE,0,0);
    sreg = sreg << 8;
    return(sreg + lsb_ch);
}

```

840

```

void load_data(unsigned char lo_byte, unsigned char hi_byte)

```

```

{
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0, SET_DLE_FLAG);
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0,hi_byte);
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0, SET_DLE_FLAG);
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0,lo_byte);
}

```

850

```

void load_addr(unsigned char lo_byte, unsigned char hi_byte)

```

```

{
    load_data(lo_byte, hi_byte);
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0, LOAD_ADDRESS); /*load RAM addr*/
}

```

860

```

void load_pc(unsigned char lo_byte, unsigned char hi_byte)

```

```

{
    load_data(lo_byte, hi_byte);
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0, WRITE_PC);
}

```

870

```

void load_psw(unsigned char lo_byte, unsigned char hi_byte)

```

```

{
    load_data(lo_byte, hi_byte);
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0, WRITE_PSW);
}

```

```
}
```

```
void CHECK_STATUS()
```

```
{                                                                                               880
    /* check status before sending data to the board or selecting... */
    /* a new node */
    for(chk_wait = 0; chk_wait < CHK_WAIT ;chk_wait++)
        {
            com1_stat = _bios_serialcom(_COM_STATUS,0,0);
            if ((com1_stat & 0x6000) == 0x6000)
                return;
        }
    printf("\nTRANSMISSION FAILED!\n");
    printf("COULD NOT COMMUNICATE WITH TARGET PROCESSOR.\n");
    printf("PLEASE CHECK HARDWARE ETC., AND RETRY.\n");
}                                                                                               890
```

```
void REC_WAIT()
```

```
{
    for(chk_wait = 0; chk_wait < CHK_WAIT ;chk_wait++)
        {
            com1_stat = _bios_serialcom(_COM_STATUS,0,0);
            if((com1_stat & 0x0100) == 0x0100)
                return;
        }
    printf("\nRECEPTION FAILED!\n");
    printf("COULD NOT COMMUNICATE WITH TARGET PROCESSOR.\n");
    printf("PLEASE CHECK HARDWARE ETC., AND RETRY.\n");
}                                                                                               900
```

```
int run_stat()
```

```
{
    /* check status to see if program running */
    com1_stat = _bios_serialcom(_COM_STATUS,0,0);
    if ((com1_stat & 0x0040) == 0x0040)
        return(1);
    else
}                                                                                               910
```

```

        return(0);
    }

/* for main menu selection 3*/
void glue_io()
{
    int data0;
    unsigned char data, i;

    _clearscreen(_GCLEARSCREEN);
    printf("\n\n\t ***** NILM Processor Selection *****\n\n");

    for(;;)
    {
        printf("\n\nEnter Processor No. (00 to 255):  ");
        scanf("%d", &data0);

        if ((data0 > 255) || (data0 < 0))
        {
            printf("\n\nInvalid Processor No.  %d\n", data0);
            continue;
        }
        data = data0;
        outp(SLAVE_NODE, data);
        cur_proc = data;
        printf("\nProcessor %d selected for communication.", data);
        printf("\n\nDo you want to Exit to Main Menu [y]?  ");
        scanf("%c", &i);
        scanf("%c", &i);
        if (i != 'n')
            return;
    }

}

/* for main menu selection 4*/
void master_io()

```

920

930

940

950

```

{

unsigned char data, i;
unsigned int j;

outp(CONT_8255, MODE_8255);

        _clearscreen(_GCLEARSCREEN);
        printf("\n\n ***** NILM MASTER BOARD INTERFACE *****\n\n");
        printf("\n\nTo EXIT: Press 'e' and ENTER.\n");
        printf("\n\nTO ACQUIRE DATA: Press any other key and then hit ENTER. ");
        scanf("%c", &i);
        scanf("%c", &i);
        if (i == 'e')
            return;

        printf("\n\n Acquiring data from Master Board...");
        get_16sec(0); /* 10240 */
        printf("\n\nData placed in files:  ch1.dat, ch2.dat,..., ch8.dat\n");
        wait_exit();
}

/* retrieve last several seconds of acquired data from master */
void get_16sec(unsigned int barrier)
{
unsigned long i;
short int reg_a = 0, reg_b = 0;
unsigned short int t = 0;
unsigned char data, reg_a_lsb, reg_b_lsb, reg_a_msb, reg_b_msb;
FILE *fopen(), *ch1, *ch2, *ch3, *ch4, *ch5, *ch6, *ch7, *ch8;

ch1 = fopen("ch1.dat", "w");
ch2 = fopen("ch2.dat", "w");
ch3 = fopen("ch3.dat", "w");
ch4 = fopen("ch4.dat", "w");
ch5 = fopen("ch5.dat", "w");
ch6 = fopen("ch6.dat", "w");
ch7 = fopen("ch7.dat", "w");
ch8 = fopen("ch8.dat", "w");

```

```

SETMODE                                     /* output PC_MODE */
while (1)                                  /* wait PC_MISC */
{
    data = inp(REGC_8255);
    if (data & MISC_MASK)
        break;
}

                                                                 1000
/*****SETFETCH*****/                       /* Assert FETCH */ /*FIRST ASSERTION*/
for (i = 0; i < 16384; i++)
{
    SETFETCH    /**TEST**/* /* Assert FETCH */
    while (1)    /* wait DAV */
        {
            data = inp(REGC_8255);
            if (data & DAV_MASK)
                break;
        }
                                                                 1010
    reg_a_lsb = inp(REGA_8255) & 0x3f;        /* fetch lsb data */
    reg_b_lsb = inp(REGB_8255) & 0x3f;
    CLRFETCH    /* Assert !FETCH */
    while (1)    /* wait !DAV */
        {
            data = inp(REGC_8255);
            if (!(data & DAV_MASK))
                break;
        }
    SETFETCH    /* Assert FETCH */
                                                                 1020
    while (1)    /* wait DAV */
        {
            data = inp(REGC_8255);
            if (data & DAV_MASK)
                break;
        }
    reg_a_msb = inp(REGA_8255) & 0x3f;        /* fetch msb data */
    reg_b_msb = inp(REGB_8255) & 0x3f;
    CLRFETCH    /* Assert !FETCH */
    while (1)    /* wait !DAV */
                                                                 1030
        {

```

```

        data = inp(REGC_8255);
        if (!(data & DAV_MASK))
            break;
    }
    /* start next point retrieval while storing current point in file */
    /**** SETFETCH ****/          /* Assert FETCH */

    if (i < barrier)
        continue;                1040

    t = 0;
    t = reg_a_msb << 2;          /* format data */
    if (t > 127)
        t+= 0xff00;
    reg_a = (t << 4) + reg_a_lsb;
    t = 0;
    t = reg_b_msb << 2;          /* format data */
    if (t > 127)
        t+= 0xff00;
    reg_b = (t << 4) + reg_b_lsb;    1050
    t = (unsigned short int) i%4;
    switch(t)
    {
        /*place data in .dat files */
        case 0:
            fprintf(ch1,"%d\n", reg_a);
            fprintf(ch5,"%d\n", reg_b);
            break;

        case 1:
            fprintf(ch2,"%d\n", reg_a);    1060
            fprintf(ch6,"%d\n", reg_b);
            break;

        case 2:
            fprintf(ch3,"%d\n", reg_a);
            fprintf(ch7,"%d\n", reg_b);
            break;

        case 3:
            fprintf(ch4,"%d\n", reg_a);    1070

```



```

        fprintf(ch8,"%d\n", reg_b);
        break;
    }
}
CLRFETCH          /* Assert !FETCH */
CLRMODE          /* output !PC_MODE */
while (1)        /* wait !MISC */
{
    data = inp(REGC_8255);
    if (!(data & MISC_MASK))
        break;
}

fclose(ch1);
fclose(ch2);
fclose(ch3);
fclose(ch4);
fclose(ch5);
fclose(ch6);
fclose(ch7);
fclose(ch8);
}

/* for main menu selection 5*/
void master_set()
{
    unsigned char data, i, m;

    outp(CONT_8255, MODE_8255);

    _clearscreen(_GCLEARSCREEN);
    printf("\n\n\t ***** MASTER MODE SET UP *****\n\n");

    if(msmode == 1)
        printf("\n** Master is in Acquisition Mode **\n");
    else
        printf("\n** Master is in Sleep Mode **\n");

    printf("\n\nSelect Function:\n\n");

```

1080

1090

1100

```

printf("1. Place Master Board in Acquisition Mode\n");
printf("2. Place Master Board in Sleep Mode\n");
printf("3. Exit \n\n");

    m = getchar();
if (m == '\n')
    m = getchar();
switch(m)
{
    case '1' :
        outp(MASTER_SET,0);
        msmode = 1;
        break;

    case '2' :
        outp(MASTER_SET,1);
        msmode = 2;
        break;

    case '3' :
        return;

}
wait_exit();
}

/* for main menu selection 6*/
void slave_set()
{
    unsigned char data, i, m;

    _clearscreen(_GCLEARSCREEN);
printf("\n\n\t ***** SLAVE MODE SET UP *****\n\n");

    printf("\n\nSelect Function:\n\n");
printf("1. Place Slave Modules in Template Acquisition Mode\n");
printf("2. Place Slave Modules in Normal Function Mode\n");
printf("3. Reset All Slave Processers\n");
printf("4. Exit \n\n");

```

1110

1120

1130

1140

```

        m = getchar();
                                                    1150
    if (m == '\n')
        m = getchar();
    switch(m)
    {
    case '1' :
        for(i = 0; i < 32; i++)
            {
                CHECK_STATUS();
                outp(SLAVE_NODE, i);
                DELAY();
                                                    1160
                load_addr(LO_MODE_LOC, HI_MODE_LOC);
                load_data(1,0);
                CHECK_STATUS();
                _bios_serialcom(_COM_SEND, 0, WRITE_BYTE);
            }
        outp(SLAVE_NODE, cur_proc);
        printf("\nSlave Modules in Template Acquisition Mode\n");
        break;

    case '2' :
                                                    1170
        for(i = 0; i < 32; i++)
            {
                CHECK_STATUS();
                outp(SLAVE_NODE, i);
                DELAY();
                load_addr(LO_MODE_LOC, HI_MODE_LOC);
                load_data(0,0);
                CHECK_STATUS();
                _bios_serialcom(_COM_SEND, 0, WRITE_BYTE);
            }
                                                    1180
        outp(SLAVE_NODE, cur_proc);
        printf("\nSlave Modules in Normal Mode\n");
        break;

    case '3' :
        rst_slave();
    break;

```

```

                case '4' :
                    return;
            }
            wait_exit();

}

/* pause for user input before exiting from submenu */
void wait_exit()
{
char ch;
    printf("\nPress Any Key To Continue...\n");
    while (!kbhit());
    ch = getch();
}

/* within menu 6, reset slaves */
void rst_slave()
{
unsigned char k;

_clearscreen(_GCLEARSCREEN);
printf("\n\n ***** RESET ALL SLAVE PROCESSORS *****\n\n");
printf("\n\n Are you sure you want to reset all slave processors [n]? ");
scanf("%c", &k);
scanf("%c", &k);
if (k != 'y')
return;

/* used within rst_slaves() and processor initialization */
reset_it();
}

void reset_it()
{
    int i,j, num_coll;
    unsigned char cont_data;
    printf("\n\n Resetting Processors....\n");

```

1190

1200

1210

1220

```

cont_data = inp(COM1_CONT); /* toggle DTR(Init) line to affect reset*/
cont_data = cont_data & 0xfe;
outp(COM1_CONT, cont_data);
cont_data = cont_data | 0x01;
for (j = 0; j < 500; j++)          /* Wait */
    for (i = 0; i < 32000; i++);
outp(COM1_CONT, cont_data);
printf("\n\nAll Processors Reset. \n");
}

void coll_comm()
{
/* This routine places the host in a communication loop with the collater */
/* slaves. It checks if there is a hit. If so, it gets the hit info, */
/* processes it for interscale lockout. It then calls upon the Master to */
/* send it the last 16 seconds of data and then calls a viewing utility. HAVE */
/* MADE THE MASTER COMM PROCEDURE INTO A SUBROUTINE SO THAT IT CAN BE DONE */
/* MANUALLY AS A MENU ENTRY OR AUTOMATICALLY BY THE PC WHEN THERE IS A HIT. */

int j, data;
unsigned char i, k, x;
FILE *fopen(), *inf;
char fname[15];

_clearscreen(_GCLEARSCREEN);
printf("\n\n ***** COLLATER PROCESSOR COMMUNICATION *****\n\n");

printf("\n\nDo you wish to perform collater initialization [n]? ");
scanf("%c", &k);
scanf("%c", &k);
if (k != 'y')
    goto afterwards;

/* make sure to add load names correctly ie. match load1 name to the actual */
/* load whose chain is coming in at hsi1, and load 2 is the load whose chain */
/* comes in at hsi2 of the collater, etc. */

/* make sure loadname.txt is present : */

```

1230

1240

1250

1260

```

if ((inf = fopen("loadname.txt", "r")) == NULL)
    {
        printf("\n\nCannot access file 'loadname.txt'\n");
        fclose(inf);
        wait_exit();
        return;
    }
    }

for(j = 0, i = 0; j < NUM_PROCS; j++) /*i is the collater serial no*/
    for(k = 0; k < 8; k++) /*j is the collater processor no*/
        {
            fscanf(inf,"%s", &packet[i][k].name);
            if(strcmp(packet[i][k].name,"end") == 0)
                {
                    if (k!=0)
                        {
                            id[i] = j;
                            printf("\nProcessor %d designated to be a collater\n", j);
                            i++;
                        }
                    break;
                }
            printf("\n%s designated load %d in collater %d",packet[i][k].name,k,j);
        }
    }

num_coll = i;

afterwards:
    printf("\n\nDo you wish to put the system in Acquisition Mode [y]? ");
    scanf("%c", &k);
    scanf("%c", &k);
    if (k == 'n')
        return;
    /* check collaters on a round_robin basis, until a hit is seen */
    armed();
}

/* continuation of coll_comm() */
void armed()

```

1270

1280

1290

1300

```

{
int j;
unsigned char i, k, x, entry;
unsigned char hits, any_hit, acqhi, acqlo;
unsigned short acq;

                                                                    1310

after_init:
_clearscreen(_GCLEARSCREEN);
printf("\n\n ***** System in Acquisition Mode ***** \n\n");
printf("\n To Exit, press any key.\n");
printf("\n\n Waiting for a load hit...");

while(!kbhit())
{
    hits = 0;
    entry = 0;                                                                    1320
    forloop:
    for(j = 0; j < num_coll; j++)        /* For each collater */
        {
            CHECK_STATUS();
            outp(SLAVE_NODE, id[j]);
            cur_proc = id[j];
            DELAY2();
            /* Iteratively retrieve the 8 load hit locations */
            /* Is there a hit? */
            /* if so, retrieve packet and put in packet [j] */
            load_addr(ADD_LO_HIT, ADD_HI_HIT);
            get_byte();
            any_hit=_bios_serialcom(_COM_RECEIVE,0,0);
            if (!any_hit)
                {
                    DELAY(); /* pause between polling */
                    continue;
                }
            DELAY2(); /* let all hits come in */
            if(entry = 0) /* if the first */
                {
                    /* then search all collaters */
                    entry = 1;
                    goto forloop;
                }
            }
                                                                    1330
                                                                    1340

```

```

}
/* get current acq. time */
load_addr(0x00,0x35);
get_byte();
acqlo=_bios_serialcom(_COM_RECEIVE,0,0);
load_addr(0x01, 0x35);
get_byte();
acqhi=_bios_serialcom(_COM_RECEIVE,0,0);
acq = acqlo + (acqhi << 8);
for (i=0, x=0; i < 8; i++, x = x + 0x10)
    {
    load_addr(x, HI_ADDR);
    get_byte();
    packet[j][i].ld_hit=_bios_serialcom(_COM_RECEIVE,0,0);
    if (packet[j][i].ld_hit == 1)
        {
        get_hit(i,j);
        packet[j][i].cur_time = (acqhi << 8) + acqlo;
        hits++;
        }
    }
load_addr(ADD_LO_HIT, ADD_HI_HIT); /* anyhit is reinitialized to 0 */
load_data(0,0);
CHECK_STATUS();
_bios_serialcom(_COM_SEND,0, WRITE_BYTE);
}
if (hits > 0) /* Tell User , update file */
    {
    printf("\n\n%d hit(s) recorded\n", hits); /*leave*/
    update(); /* contacts.txt */
    history(); /* history.txt */
    /* reset hit record */
    for(i = 0; i < num_coll; i++) /*leave*/
        for(j= 0; j < 8; j++) /*leave*/
            packet[i][j].ld_hit = 0; /*leave*/
    printf("\nRetrieving last 12 sec. Data from Master Board...");
    get_16sec(0); /* 6144 */
    printf("Done\n\n");
    mark();

```



```

        i = getch();
        goto after_init;
    }
}
i = getch();
printf("\n\nNo longer in Collater Acquisition Mode.\n");
wait_exit();
}

```

1390

```

/* subroutine for retriving hit info data */
void get_hit(unsigned char load, int coll)
{
    int i;
    unsigned char base;

    base = (load*0x10);
    load_addr(base,0x30);    /* send address */
    load_data(0,0);    /* ld.hit is reinitialized to 0 */
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0, WRITE_BYTE);

    base+= 3;
    load_addr(base,0x30);    /* send address */
    get_byte();
    packet[coll][load].load_id= _bios_serialcom(_COM_RECEIVE,0,0);
    get_byte();
    packet[coll][load].scale= _bios_serialcom(_COM_RECEIVE,0,0);
    get_byte();
    packet[coll][load].hi_loc= _bios_serialcom(_COM_RECEIVE,0,0);
    get_byte();
    packet[coll][load].lo_loc= _bios_serialcom(_COM_RECEIVE,0,0);
    get_byte();
    packet[coll][load].hi_time= _bios_serialcom(_COM_RECEIVE,0,0);
    get_byte();
    packet[coll][load].lo_time= _bios_serialcom(_COM_RECEIVE,0,0);
}

```

1400

1410

1420

```

void get_byte()
{
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0, READ_BYTE);
    CHECK_STATUS();
    _bios_serialcom(_COM_SEND,0, TRANSMIT);
    REC_WAIT();
}
1430

/* update contacts in contacts.txt */
void update()
{
    unsigned int i, j, loc, time;
    FILE *fopen(), *outf;
    outf = fopen("contacts.txt", "w");
    for(i = 0; i < num_coll; i++)
    for(j= 0; j < 8; j++)
    {
        if(packet[i][j].ld_hit)
            1440
            {
                loc= (packet[i][j].hi_loc<<8)+packet[i][j].lo_loc;
                time= (packet[i][j].hi_time<<8)+packet[i][j].lo_time;
                fprintf(outf,"=====\n");
                fprintf(outf,"Load:='%s'\n", packet[i][j].name);
                fprintf(outf,"Time:=%d\n", time);
                /*fprintf(outf,"Location:=%d\n",loc);*/
                fprintf(outf,"Scale:=%d\n", packet[i][j].scale);
            }
    }
    1450
    fclose(outf);
}

/*update history.txt with latest hit */
void history()
{
    unsigned int i, j, loc, time;
    FILE *fopen(), *outf;
    outf = fopen("history.txt", "a");
    fprintf(outf,"\n*****");
    1460
}

```

```

    for(i = 0; i < num_coll; i++)
        for(j= 0; j < 8; j++)
            {
                if(packet[i][j].ld_hit)
                    {
                        loc= (packet[i][j].hi_loc<<8)+packet[i][j].lo_loc;
                        time= (packet[i][j].hi_time<<8)+packet[i][j].lo_time;
                        fprintf(outf,"\n\nLoad:  '%s'\n", packet[i][j].name);
                        fprintf(outf,"Time:  %d\n", time);
                        /*fprintf(outf,"Location: %d\n",loc);*/
                        fprintf(outf,"Scale:  %d\n", packet[i][j].scale);
                    }
            }
        fclose(outf);
}

/* delay used to allow slave selection etc. to go through; also used for */
/* delay between successive polling of collaters */
void DELAY()
{
    long int i;
    for (i = 0; i < 30000; i++);
}

/* delay used in collater communication to wait until all hits are */
/* received by the collater */
void DELAY2()
{
    long int i,j;
    for (i = 0; i < 100000; i++)
        for (j = 0; j < 100; j++);
}

```

E.2 Modifications Made to 80C196KC RISM

As stated in chapter 5, the RISM for MLM is a modified version of the RISM developed for the 80C196KC Evaluation Board. The main differences arise from the fact that in the MLM, the PC communicates with the slaves via the P2-2 interrupt, not the NMI, as in the evaluation board setup. Assembly Code for the 80C196KC RISM is given in Appendix C in [14], complete with remarks and line numbers. The instruction set for the 80C196KC is given in [13], and [22] discusses the assembler for the 80C196 family. The code in [14] is referenced directly in the listing that follows of the changes made:

Line 277 is changed from:

```
0047 277 external_int: dcw 4700H ;P2-2 INT vector= 4700H
```

to:

```
0000 277 external_int: dcw 0000H ;P2-2 INT vector= 0000H
```

Line 336 is changed from:

```
A1000036 336 ld tempw, #rism_psw ;rism_psw = 0000
```

to:

```
A1800236 336 ld tempw, 0x0280 ;NEW VALUE FOR PSW
```

Line 340 is changed from:

```
1136 340 clrb char ;clear byte
```

to:

```
0136 340 clr w tempw ;clear word
```

Line 391 is changed from:

```
A1000036 391 ld tempw, #rism_psw ;rism_psw = 0000
```

to:

```
A1800236 391 ld tempw, 0x0280 ;NEW VALUE FOR PSW
```

line 403 is changed from:

```
27F5 403 br diag_pause_loop ;to diag_pause_loop
```

to:

205B 403 br #218BH ;to nearest RISM_EXIT

line 481 is changed from:

C92221 481 push #(diag_pause_offset) ;to diag_pause_loop

to:

C90410 481 push #1004H ;to nearest RISM_EXIT

Line 482 is changed from:

C90000 482 push #rism_psw ;rism_psw = 0000

to:

C98002 482 push 0x0280 ;NEW VALUE FOR PSW

Line 676 is changed from:

A1000036 676 ld tempw, #rism_psw ;rism_psw = 0000

to:

A1800236 676 ld tempw, 0x0280 ;NEW VALUE FOR PSW

Line 684 is changed from:

C3013E2000 684 st zero, (nmi_offset)[0] ;initialize nmi

to:

C3010E2000 684 st zero, (p2-2_offset)[0] ;initialize P2-2

line 687 is changed from:

27FE 687 br monitor_pause ;to monitor_pause

to:

26E2 687 br #1C00H ;to new code at 1C00

The following code is added to the RISM at location 1C00:

1C00H: C98002 push 0x0280 ;NEW VALUE FOR PSW

1C03H: F3 popf ;store value of 0280 in PSW

1C04: 27FE br 1C04 ; wait here for command

Line 727 is changed from:

C9201D 727 push #(monitor_pause_offset) ;to monitor_pause_loop

to:

C9041C 727 push #1C04 ;to nearest RISM_EXIT

Line 728 is changed from:

C90000 728 push #rism_psw ;rism_psw = 0000

to:

C98002 728 push 0x0280 ;NEW VALUE FOR PSW

Appendix F

PC I/O Card for MLM Control

As described in Chapter 5, a PC I/O card was used to implement slave module selection and PC communication with the master board. The JDR PR-2 prototype card was the I/O card of choice. The PR-2 is an 8-bit I/O card with prototyping space available for housing application circuitry [23].

The schematics for the card, as given in [23], are shown in Figure F-1. All PC-AT data, address and control signals used are buffered by IC1..3, so that the PC Bus is protected from errors in the prototype circuit constructed on the I/O card. A 4 position DIP Switch and a comparator (IC6) use address lines A[8..5] to place the I/O card in the PC's memory map [24], [25]. The address range for which the board will be selected is 0x300–0x31F. Address lines A[4..2] are decoded by an LS138 (IC5 in Figure F-1) to provide eight Chip Select lines [28]. Each Chip Select output from the LS138 will enable four consecutive addresses. For instance, pin 14 (Select 1) will be active (low) for Addresses 0x304–0x307.

These blocks of four addresses are especially useful for our prototype circuit as it used the 8255 General Purpose I/O IC, which has a bank of four registers [16]. The 8255 was used to communicate with the master board. Its Control Register is initialized with the control byte needed to configure its three ports as follows: Port A receives data from input streams 1–4, Port B receives data from input channels 5–8 and Port C is used in the following manner to accomplish the handshaking protocol:

PortC.0: Inputs the DAV2PC signal from the Master Board PALs.

PortC.1: Inputs the PC_MISC_ACK signal from the Master Board PALs.

PortC.4: Outputs the PC_MODE signal to the Master Board PALs.

PortC.5: Outputs the PC_FETCH signal to the Master Board PALs.

An octal register, the LS574, is dedicated to slave node selection. Its inputs are connected to the buffered 8 LSBs of the Data Bus. Its \overline{OE} pin is permanently enabled (tied low). Its CLK pin (clock input) is tied to a Select line from the decode circuitry on the I/O card. If a memory write is executed at the address to which this register is mapped (via the I/O card circuitry), the register will be selected and the pulse on its CLK input will cause data on the bus to be loaded in. The outputs of this register are connected via ribbon cables to the glue logic circuitry on each slave board. ECM can thus select a slave module by writing the ID byte to the memory address of this register.

Another LS574 is used to set the master board mode. Its least significant input line accepts the buffered LSB of the Data Bus. Its \overline{OE} pin is also permanently enabled (tied low). Its CLK pin (clock input) is tied to another Select line from the decode circuitry on the I/O card. Its LSB output bit is connected directly to an input pin of PAL_TR2, one of the *Transfer PALs* on the master board. ECM sets the master board mode by writing a 0 (acquisition mode) or a 1 (sleep mode) to this register's address in PC memory.

The addresses in PC memory where each of these registers resides are defined in the ECM software as follows:

```
#define IOBASE_ADDR 0x300
#define REGA_8255 (IOBASE_ADDR + 0x00)
#define REGB_8255 (IOBASE_ADDR + 0x01)
#define REGC_8255 (IOBASE_ADDR + 0x02)
#define CONT_8255 (IOBASE_ADDR + 0x03)
#define SLAVE_NODE (IOBASE_ADDR + 0x04)
#define MASTER_SET (IOBASE_ADDR + 0x08)
```

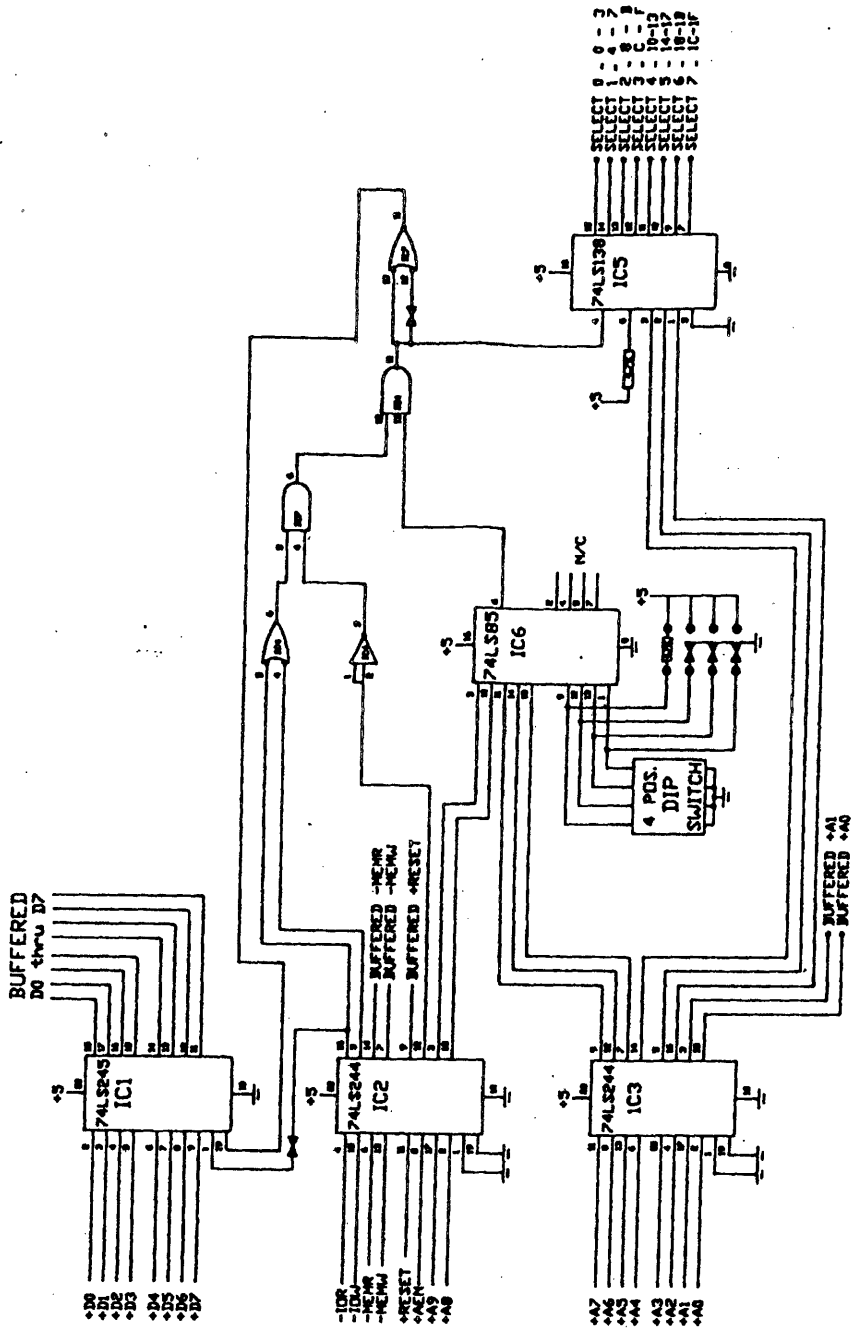



Figure F.1: PC I/O Board Schematics

Appendix G

The Analog Preprocessor

In this appendix we include the paper titled “Harmonic Estimates for Transient Event Detection” as presented at the Universities Power Engineering Conference (UPEC) in 1994. This describes the salient features of the Analog Preprocessing Board used in the MLM setup and aids our understanding of the nature of the MLM input streams.

Harmonic Estimates for Transient Event Detection

Steven B. Leeb Steven R. Shaw

Abstract

This paper describes a preprocessor that computes, for an observed waveform, the spectral envelopes associated with a time-varying Fourier series. Spectral envelopes have proven remarkably useful for transient event detection and identification at the utility service entry of a building. The performance of the preprocessor is illustrated with results from a prototype.

Background

The transient behavior of a typical electrical load is strongly influenced by the physical task that the load performs. The load survey conducted in [1] indicates that nonlinearities

in the constitutive laws of the elements that comprise a load, or in the state equations that describe a load, or both, tend to create interesting and repeatably observable turn-on transient profiles suitable for identifying specific load classes. The turn-on transients associated with a fluorescent lamp and an induction motor, for example, are distinct because the physical tasks of igniting an illuminating arc and accelerating a rotor are fundamentally different. Transient profiles tend not to be eliminated even in loads which employ active waveshaping or power factor correction.

This observation has led to the development of a transient event detector for non-intrusive load monitoring. The nonintrusive load monitor (NILM) determines the operating schedule of the major electrical loads in a building from measurements made solely at the utility service entry [2], [3]. For electric utilities and industrial facilities managers, the NILM is a convenient and economical means of acquiring accurate energy usage data with a minimal installation effort. In [1] and [4], a multiscale transient event detection algorithm was introduced that can identify individual loads operating in a building by examining measured transient profiles observed in the aggregated current waveforms available at the service entry. This detection algorithm extends the applicability of the NILM to demanding commercial and industrial sites, where substantial efforts, e.g., power factor correction and load balancing, are made to homogenize the steady state behavior of different loads, and where loads may turn on and off very frequently.

With the incorporation of the transient event detector, the NILM is also a potentially important platform for power quality monitoring and for monitoring the performance of critical loads. For example, the NILM can track down power quality offenders, i.e., loads which draw extremely distorted, non-sinusoidal input current waveforms, by correlating the introduction of undesired harmonics with the operation of specific loads at a target site. The performance of the detection algorithm has been demonstrated with results from a prototype real-time event detector implemented with a digital signal processor [1], [4].

A critical observation made during the development of the prototype is that direct examination of a current waveform at the service entry, or a closely related waveform like instantaneous power, may not reveal key features for transient identification. It is essential to isolate key features from near constant frequency, "carrier wave" type signals like 120 Hz instantaneous power so that slight errors in matching carrier frequency with a template do not dominate the results of a recognition system searching for a modulating envelope. A preprocessor in the prototype event detector eliminates carrier frequency artifacts from input data by averaging over at least one carrier wave period to generate a short time estimate of spectral content. The slow envelopes of the windowed time average of instantaneous power, i.e., real power, and of reactive power, and of higher harmonic content, are found by mixing the observed current with appropriate sinusoids and then low-pass filtering. This paper describes an advanced preprocessor, or spectral envelope estimator, for use in the prototype event detector.

Spectral Envelope Estimation

The development of the spectral envelope estimator is stimulated by the generalized averaging techniques presented in [5] and by the short time Fourier transform and Fourier series methods presented, for example, in [6] and [7] for speech processing and power systems simulation, respectively. With minor restrictions which cause no limitations in a practical power systems setting, a waveform $x(\tau)$ given as a function of τ may be described with

arbitrary accuracy at time $\tau \in (t - T, t]$ by a Fourier series with *time-varying*, complex spectral coefficients $a_k(t)$ and $b_k(t)$:

$$x(t - T + s) = \sum_k a_k(t) \cos(k \frac{2\pi}{T}(t - T + s)) + \sum_k b_k(t) \sin(k \frac{2\pi}{T}(t - T + s)) \quad (G.1)$$

The variable k ranges over the set of non-negative integers; T is a real period of time, and $s \in (0, T]$.

The coefficients $a_k(t)$ may be found from the formula [5], [8]:

$$a_k(t) = \frac{2}{T} \int_0^T x(t - T + s) \cos(k \frac{2\pi}{T}(t - T + s)) ds \quad (G.2)$$

Similarly, the coefficients $b_k(t)$ are computed by the formula:

$$b_k(t) = \frac{2}{T} \int_0^T x(t - T + s) \sin(k \frac{2\pi}{T}(t - T + s)) ds \quad (G.3)$$

In practice, Eqns. G.2 and G.3 can be used to compute the evolution in time of the spectral coefficients $a_k(t)$ and $b_k(t)$ as an interval of interest of width T slides over the waveform x . The coefficients $a_k(t)$ and $b_k(t)$ as functions of time will be referred to as the *spectral envelopes* of x for the harmonic k .

Estimates of the spectral envelopes of current waveforms observed at the utility service entry of a building have proven remarkably useful for transient event detection in the NILM, for at least two reasons. First, even for waveforms x with substantial high frequency content, the frequency content of the spectral envelopes can be made relatively band-limited. As will be seen, this tends to ease the sample rate requirements on any single channel of the transient event detector. Second, in steady state operation especially, estimates of the spectral envelopes serve as direct indicators of real and reactive power, as well as potentially undesirable harmonic content. Demonstrations of these claims follow.

For convenience, let

$$x_c(t) = \frac{2}{T} x(t) \cos(k \frac{2\pi}{T} t)$$

and

$$x_s(t) = \frac{2}{T} x(t) \sin(k \frac{2\pi}{T} t).$$

represent sinusoids modulated by the function x . Equations G.2 and G.3 are equivalent to convolving in time the integrands $x_c(t)$ and $x_s(t)$, respectively, with a rectangular pulse $p(t)$ with unit height extending from time 0 to time T , and may be written as

$$a_k(t) = x_c(t) \otimes p(t) \quad (G.4)$$

and

$$b_k(t) = x_s(t) \otimes p(t) \quad (G.5)$$

where the symbol \otimes represents convolution operator. In the frequency domain, the contin-

uous time Fourier transform of the spectral envelope $a_k(t)$ is

$$A_k(f) = \int_{-\infty}^{\infty} a_k(t) e^{-j2\pi ft} dt$$

From Eq. G.4, the magnitude of $A_k(f)$ is equivalent to the product of the magnitudes of the functions $X_c(f)$, the Fourier transform of $x_c(t)$, and $P(f)$, the Fourier transform of the pulse $p(t)$. Similarly from Eq. G.5, the magnitude of $B_k(f)$, the continuous time Fourier transform of $b_k(t)$, is the product of the magnitudes of $X_s(f)$, the transform of $x_s(t)$, and $P(f)$.

The effect of computing the spectral coefficients as an integral or average over the interval T is to attenuate the high frequency content of the spectral envelopes. Equivalently, the high frequency content of the spectral envelopes is attenuated by the (roughly) low-pass character of $P(f)$. The localization or high frequency attenuation in the frequency content of the spectral envelopes increases as the interval T increases in extent.

Each spectral coefficient indicates as a function of time the relative contribution of a sinusoid in the summations of Eq. G.1. By varying the interval T it is possible to restrict to an essentially arbitrary degree the frequency content of the spectral envelopes, regardless of the harmonic k under consideration. In an actual implementation of a transient event detector, a decomposition of even a relatively broad-band waveform x into spectral envelopes permits a trade-off, therefore, between sample rate per data channel and the number of data channels employed.

A second advantage of examining a waveform x in terms of spectral envelopes is the correspondence of the coefficients to familiar physical quantities in steady state. The term “steady state” is here taken to refer to a waveform or section of a waveform that is periodic. The interval T is presumed to be a positive integer multiple of the fundamental period of this waveform. Consider, for example, the situation in which the waveform x corresponds to an observed current waveform on a single phase of a utility system with a sinusoidal voltage waveform. For purposes of illustration, consider the voltage to be a cosine with angular frequency $2\pi/T$. Intuitively, Eqns. G.2 and G.3 compute the spectral coefficients by demodulating the periodic waveform x with an appropriate, harmonic sinusoid and low-pass filtering to preserve only the resulting lowest frequency components. For a periodic current waveform x with period T , the spectral coefficient $a_1(t)$ corresponds to a quantity that is proportional to the conventional definition of real or “time average” power [9]. Similarly, the coefficient $b_1(t)$ is proportional to reactive power. Higher order spectral coefficients correspond to in-phase and quadrature harmonic component content as in a conventional Fourier series decomposition of a periodic waveform.

Without attempt at a mathematical exposition, we observe in passing that even for waveforms which are not strictly periodic, spectral envelopes may have appealing interpretations as quantities like real and reactive power, even though such quantities are not strictly and universally defined for transient or non-periodic waveforms. For waveforms that satisfy the “slowly varying magnitude and phase” arguments commonly used to justify quasi-sinusoidal steady state approximations [7], the spectral envelopes can be loosely interpreted as the slowly varying envelopes of real and reactive power, and of harmonic content.

While not necessarily essential, the ability to associate spectral envelopes with physical quantities is comforting when employing spectral envelopes as “fingerprints” in a transient event detector. Variations in real and reactive power and harmonic content tend to be

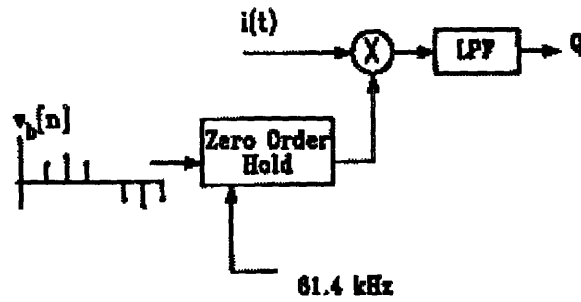


Figure G.1: Signal Flow Graph

closely linked to the physical task or energy conversion process being performed by a load. Load classes that perform physically different tasks are therefore likely to be distinguishable based on their transient behavior [1]. Since the spectral envelopes tend to be closely linked to telltale physical quantities, they are likely, and appear in practice, to serve as reliable metrics for identifying loads.

Envelope Preprocessor

The prototype transient event detector consists of two components: a preprocessor which computes estimates of the spectral envelopes, and a digital signal processing card that monitors the envelopes and runs the transient event detection algorithm. Details of the event detection algorithm have been presented in [1] and [4]. This section reviews the design of a hardware implementation of a spectral envelope preprocessor for use in the transient event detector.

To maximize flexibility, utility, and accuracy, while minimizing cost, the preprocessor computes an estimate that approximates the spectral envelope integrals of Eqns. G.2 and G.3. Figure G.1 illustrates the computation implemented in a single channel of the spectral envelope preprocessor. An observed current waveform $i(t)$, equivalent to the waveform x in the previous section, is mixed with a continuous time “staircase” or basis sinusoid. This basis sinusoid is constructed from discrete time samples, $v_b[n]$, of a desired waveform sampled at high frequency. Analytically, these samples are reconstituted into a continuous time waveform by a zero order hold (ZOH). This process is line-locked to the observed voltage waveform at the utility service entry, so that the reconstituted, basis sinusoid will exhibit a precise, desired phase with respect to the line voltage. The product of the current and basis sinusoid corresponds to a function such as $x_c(t)$ or $x_s(t)$ for a particular harmonic k , as described in the previous section. This product is low-pass filtered to yield an estimate of a particular spectral envelope for the current.

Figure G.2 shows a partial block diagram of the hardware implementation. A multiplying digital-to-analog converter (MDAC) provides the ZOH and multiplication operations shown in Fig. G.1. The observed current waveform $i(t)$ is the analog reference for the MDAC. A memory on the preprocessor contains 1024 two byte samples of the desired basis waveforms for 16 different channels or spectral envelope estimates. These samples are fed to appropriate MDAC channels under the control of the steering logic. Although only four

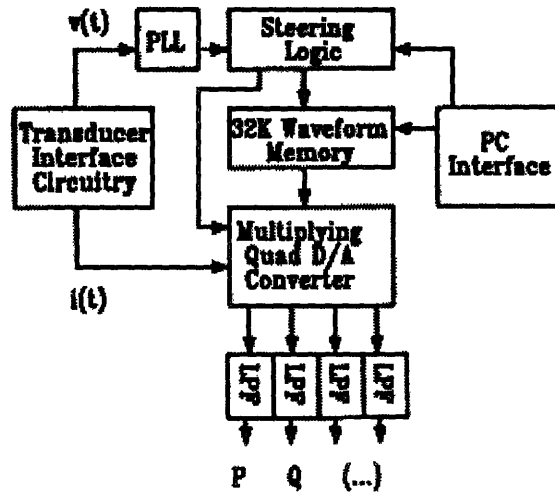


Figure G.2: Envelope Preprocessor Block Diagram

output channels are shown in Fig. G.2, a total of 16 channels are available on the prototype envelope preprocessor. A phase-locked loop (PLL) ensures that the operation of the preprocessor is synchronized to the line voltage waveform. For enhanced flexibility, the basis sample memory may be implemented either with a read-only memory, or with a static RAM which can be loaded with samples from a personal computer.

In the prototype, a second order Butterworth filter with a breakpoint at 20 Hz is used on each channel to provide an estimate of the average or low-frequency component of each MDAC output. This low-pass filter, convenient from an implementation standpoint, is obviously not identical to the windowed mean employed in Eqns. G.2 and G.3 to compute the spectral envelopes. For this reason, and because the basis waveforms are reconstructed with a (generally negligible) quantization error, the outputs of the prototype are *estimates* of the spectral envelopes. By varying the filter breakpoint, it is again possible to trade localization in time versus localization in frequency, as was possible in the previous section by varying the interval T .

Prototype Testing and Performance

The prototype spectral envelope estimator has been used in conjunction with a digital signal processor to test the transient event detection algorithm described in [4]. As an example of the envelope estimator in operation, Fig. G.3 shows the measured current waveform and some associated spectral envelopes estimated by the preprocessor during the turn-on transient of a personal computer. The top trace in Fig. G.3 shows the current into the computer during the transient. The switching power supply inside the computer initially draws a few large pulses of current as the internal bus capacitor charges from the line through a full bridge rectifier. When the capacitor has built up a substantial stored charge, the current waveform becomes "spikey" as charging begins to occur only near the peaks in the magnitude of the line voltage waveform. Approximately 0.12 seconds into the transient, the computer monitor turns on, increasing the total steady state current drawn

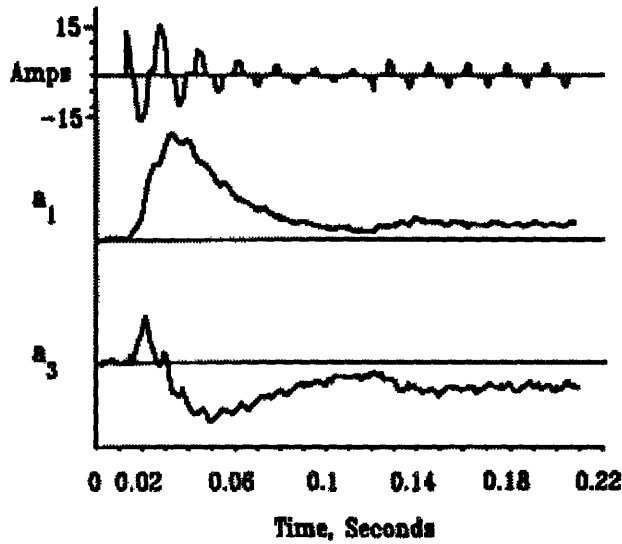


Figure G.3: Personal Computer Turn-On Transient

by the computer and monitor.

The second and third traces from the top in Fig. G.3 show two spectral envelope estimates computed by the preprocessor. The trace labeled a_1 corresponds to an envelope computed by mixing the observed current waveform with a sinusoid with the same phase and frequency as the line voltage. This trace corresponds to the slow envelope of “real power.” The second trace, labeled a_3 , indicates the spectral envelope of in-phase third harmonic content. As might be intuitively expected from examining the “spikey” line current waveform, the computer draws a substantial third harmonic current in steady state.

The spectral envelopes for different loads, such as those for the computer shown in Fig. G.3, have proven remarkably useful and robust for performing transient event detection [1], [4].

Acknowledgements

This research was funded by EPRI and ESEERCo. The authors gratefully acknowledge the valuable advice and support of Professors James Kirtley, Jr., George Verghese, and Les Norford, Dr. Richard Tabors, and Laurence Carmichael.

Bibliography

- [1] Leeb, S.B., "A Conjoint Pattern Recognition Approach to Nonintrusive Load Monitoring," M.I.T. Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, February 1993.
- [2] Tabors, R., et. al., "Residential Nonintrusive Appliance Load Monitor," EPRI final report, to appear.
- [3] Hart, G.W., "Nonintrusive Appliance Load Monitoring," *Proceedings of the IEEE*, Vol. 80, No. 12, pp. 1870–1891, December 1992.
- [4] Leeb, S.B., J.L. Kirtley, Jr., "A Multiresolution Transient Event Detector for Nonintrusive Load Monitoring," *International Conference on Industrial Electronics, Control, and Instrumentation 1993*, Maui, Hawaii, pp. 354–359, November 1993.
- [5] Sanders, S.R., J.M. Noworolski, X.Z. Liu, and G.C. Verghese, "Generalized Averaging Method for Power Conversion Circuits," *IEEE Transactions on Power Electronic Circuits*, Vol. 6, No.2, pp. 251–259, April 1991.
- [6] Oppenheim, A.V., *Applications of Digital Signal Processing*, Prentice-Hall, Inc., Engelwood Cliffs, New Jersey, pp. 117–168.
- [7] DeMarco, C.L., and G.C. Verghese, "Bringing Phasor Dynamics into the Power System Load Flow," *North American Power Symposium*, 1993.
- [8] W.M. Siebert, *Circuits, Signals, and Systems*, McGraw-Hill, New York, New York, p. 370.
- [9] Kirtley, Jr., J.L., "AC Power Flow in Linear Networks," M.I.T. 6.061 Supplementary Notes 2, June 1987.

Appendix H

Miscellaneous Details of Prototype Testing

This appendix supplements the discussion of the MLM prototypes' performance, in Chapter 6. It lists the v-section sets chosen to represent the loads monitored by the two MLM prototypes. It also includes the software routines (implemented on the PC controlling the circuit breaker panel) used to activate multiple loads at about the same time.

H.1 V-section Sets

Figures 6.2...6.7 in Chapter 6 show the transients for the six loads used in prototype testing. V-sections for these loads are listed below:

Small Motor (4 VSs): Rising and falling edges in P and Q (Fig. 6.2).

Rapid (4 VSs): Rising edge of central hump in P, step at the end of P transient, rising edge in Q in initial spike, and step down in 3P (Fig. 6.3).

Instant (2 VSs): Spike in P and the step in 3P (Fig. 6.4).

Light (1 VS): Spike in P (Fig. 6.5).

Computer (2 VSs): Spike in P and the upward step in 3P (Fig. 6.6).

Big Motor (4 VSs): Rising and falling edges in P and Q, on scale 2 (Fig. 6.7).

H.2 Software for Multiple Load Activation

Three routines are included here as examples of the software written to generate sequences of load turn-ons, used to challenge the MLM prototypes. The first file turns on two loads, the second activates three loads and the final listing turns three loads on in succession.

```
/****** UMAIR85.C *****/
/* Turns on: sm. motor (8) + rapid5 (or light5) */

#include<stdio.h>

main()
{
    long i,l,l2;

    printf ("Enter a delay interval: "); /* get delay1 from user */
    scanf("%ld",&l);

    outp(512,128);      /*Turn on first load */
    for(i = 0; i < l; i++);
    outp(512,16+128);   /*Turn on second load */
    for(i = 0; i < 400000; i++);
    outp(512,0);       /*Turn off all loads */
}
```

```
/****** UMAIR582.C *****/
/* Turns on: rapid- or light-(5) + motor(8) + instant(2)*/

#include<stdio.h>

main()
{
    long i,l,l2;

    printf ("Enter a delay interval: "); /* get delay1 from user */
    scanf("%ld",&l);

    printf ("Enter a delay interval2: "); /* get delay2 from user */
    scanf("%ld",&l2);
```

```

outp(512,16);          /*Turn on first load */
for(i = 0; i < l; i++);
outp(512,128+16);     /*Turn on second load */
for(i = 0; i < l2; i++);
outp(512,2+16+128);   /*Turn on third load */
for(i = 0; i < 60000; i++);
outp(512,0);          /*Turn off all loads */
}

```

20

```

/***** UM7852.C *****/
/* Turns on: big motor(7) + sm. motor(8) + rapid-or light-(5) + instant(2)*/

```

```

#include<stdio.h>

```

```

main()

```

```

{

```

```

    long i,l,l2,l3;

```

```

    printf("Enter a delay interval: "); /* get delay1 from user */

```

10

```

    scanf("%ld",&l);

```

```

    printf("Enter a delay interval2: "); /* get delay2 from user */

```

```

    scanf("%ld",&l2);

```

```

    printf("Enter a delay interval3: "); /* get delay3 from user */

```

```

    scanf("%ld",&l3);

```

```

    outp(512,64);          /*Turn on first load */

```

```

    for(i = 0; i < l; i++);

```

```

    outp(512,128+64);     /*Turn on second load */

```

```

    for(i = 0; i < l2; i++);

```

20

```

    outp(512,16+128+64);   /*Turn on third load */

```

```

    for(i = 0; i < l3; i++);

```

```

    outp(512,2+128+16+64); /*Turn on fourth load */

```

```

    for(i = 0; i < 40000; i++);

```

```

    outp(512,0);          /*Turn off all loads */

```

```

}

```

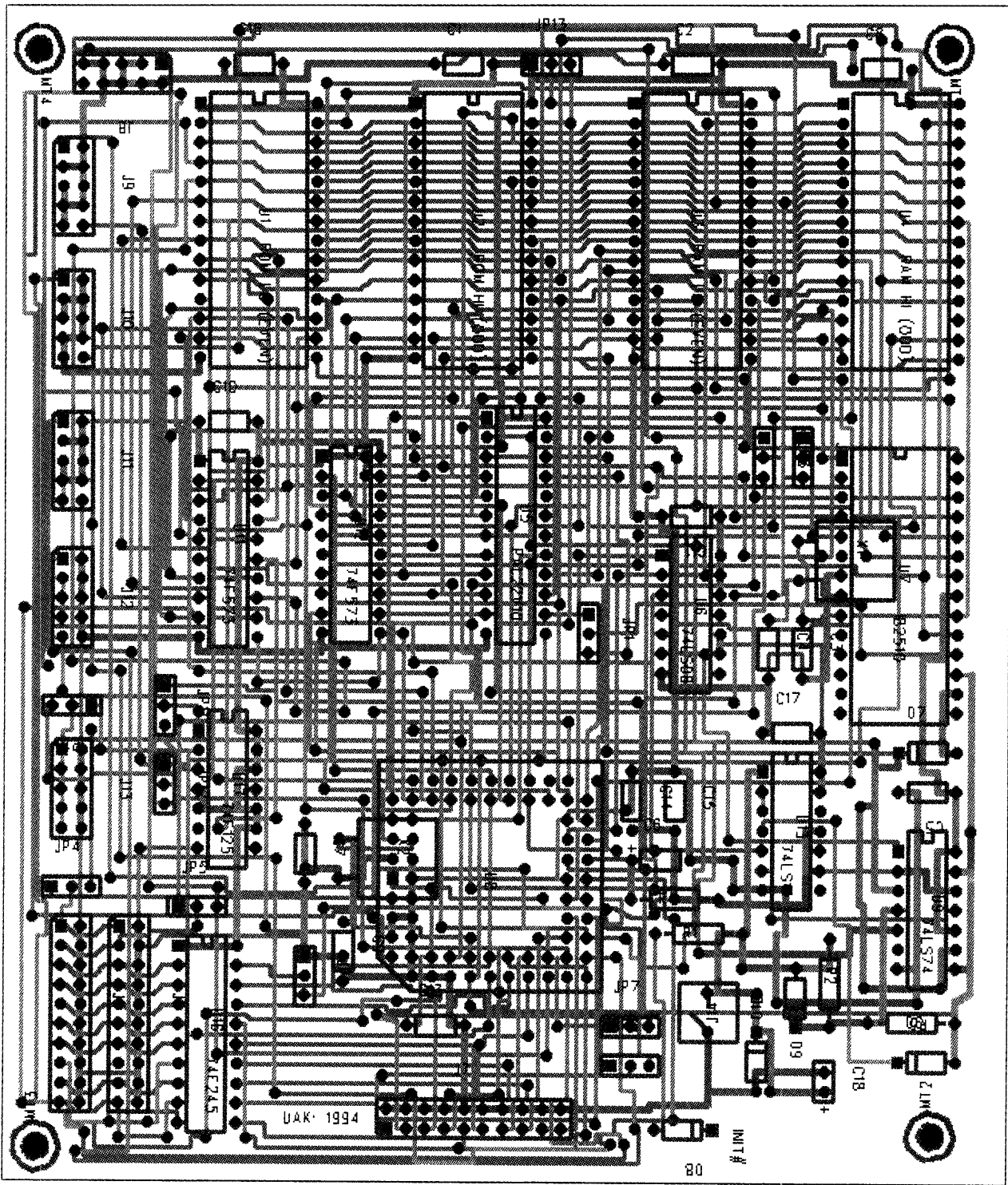
Appendix I

The Genesis of the MLM

We end this thesis with a brief pictorial guide to the making of the MLM prototypes. Seven color photographs/printouts capture the various stages along the MLM's metamorphosis. They are listed below in order of appearance:

- Figure I.1: The slave module is conceived on PADS. (Note that the figure is not captioned).
- Figure I.2: A slave board is configured as part of MLM-16S.
- Figure I.3: The MLM-16S during testing phase.
- Figure I.4: The MLM-16S all ready for load monitoring (at last!).
- Figure I.5: The final setup with both MLM prototypes¹ and the Host PC in the cabinet, the analog preprocessor to the right, and the computer controlling the circuit breaker panel, at extreme right.
- Figures I.6 and I.7: Two instances of the MLM at work, as seen in living color.

¹The bigger prototype, MLM-16S is at the bottom of the cabinet. Note the photo and the inscription which represent the nickname given to MLM-16S: **ANIMAL** (Advanced NonIntrusive Monitor for Arbitrary Loads).



UAK-1994

80

INT4

INT2

INT1

INT3

INT3

INT1

J1

J2

J3

J4

J5

J6

J7

J8

J9

J10

J11

J12

J13

J14

J15

J16

J17

J18

J19

J20

J21

J22

J23

J24

J25

J26

J27

J28

J29

J30

J31

J32

J33

J34

J35

J36

J37

J38

J39

J40

J41

J42

J43

J44

J45

J46

J47

J48

J49

J50

J51

J52

J53

J54

J55

J56

J57

J58

J59

J60

J61

J62

J63

J64

J65

J66

J67

J68

J69

J70

J71

J72

J73

J74

J75

J76

J77

J78

J79

J80

J81

J82

J83

J84

J85

J86

J87

J88

J89

J90

J91

J92

J93

J94

J95

J96

J97

J98

J99

J100

J101

J102

J103

J104

J105

J106

J107

J108

J109

J110

J111

J112

J113

J114

J115

J116

J117

J118

J119

J120

J121

J122

J123

J124

J125

J126

J127

J128

J129

J130

J131

J132

J133

J134

J135

J136

J137

J138

J139

J140

J141

J142

J143

J144

J145

J146

J147

J148

J149

J150

J151

J152

J153

J154

J155

J156

J157

J158

J159

J160

J161

J162

J163

J164

J165

J166

J167

J168

J169

J170

J171

J172

J173

J174

J175

J176

J177

J178

J179

J180

J181

J182

J183

J184

J185

J186

J187

J188

J189

J190

J191

J192

J193

J194

J195

J196

J197

J198

J199

J200

J201

J202

J203

J204

J205

J206

J207

J208

J209

J210

J211

J212

J213

J214

J215

J216

J217

J218

J219

J220

J221

J222

J223

J224

J225

J226

J227

J228

J229

J230

J231

J232

J233

J234

J235

J236

J237

J238

J239

J240

J241

J242

J243

J244

J245

J246

J247

J248

J249

J250

J251

J252

J253

J254

J255

J256

J257

J258

J259

J260

J261

J262

J263

J264

J265

J266

J267

J268

J269

J270

J271

J272

J273

J274

J275

J276

J277

J278

J279

J280

J281

J282

J283

J284

J285

J286

J287

J288

J289

J290

J291

J292

J293

J294

J295

J296

J297

J298

J299

J300

J301

J302

J303</

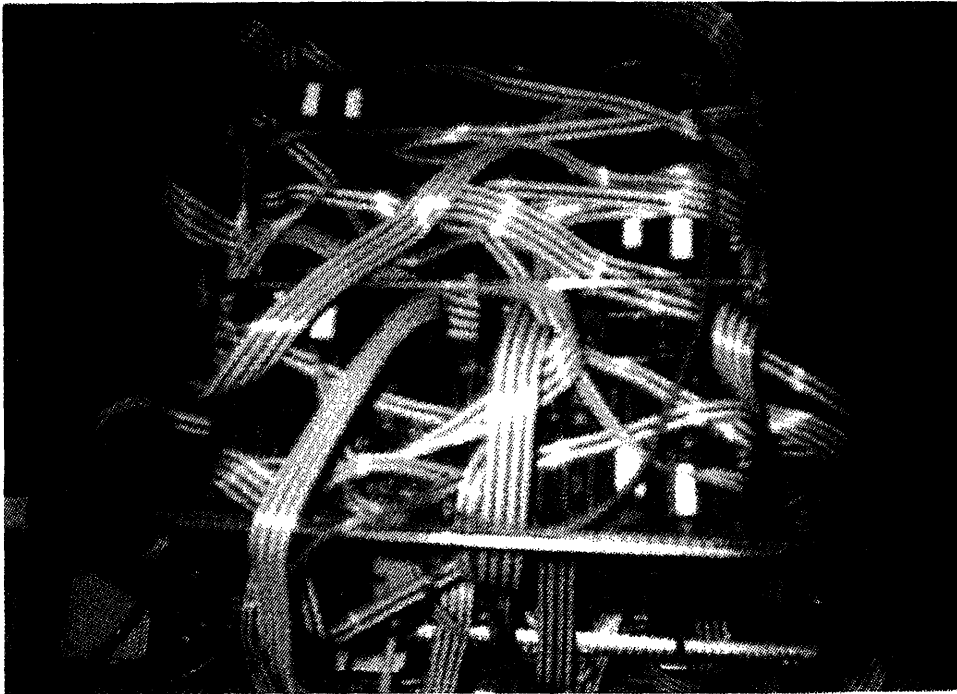


Figure I.2: A Slave Board is Configured as Part of MLM-16S

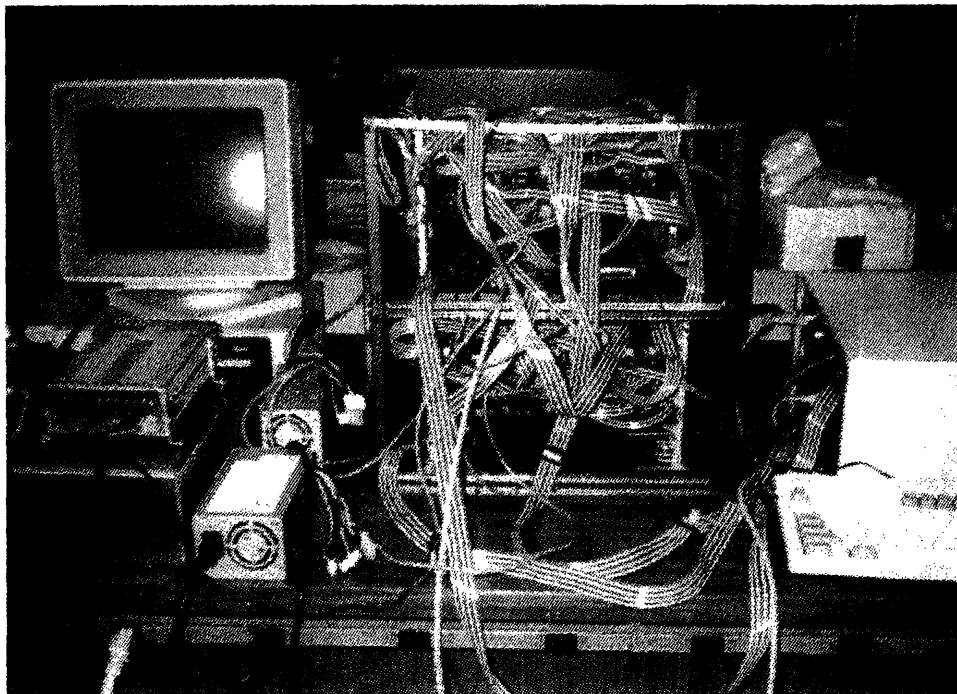


Figure I.3: The MLM-16S During the Testing Phase

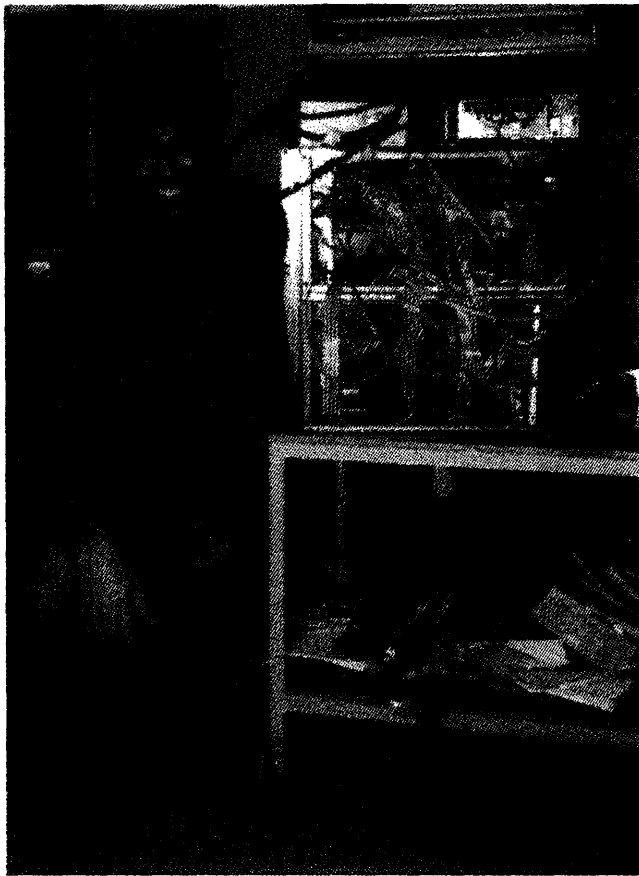


Figure I.4: The MLM-16S all Ready for Load Monitoring

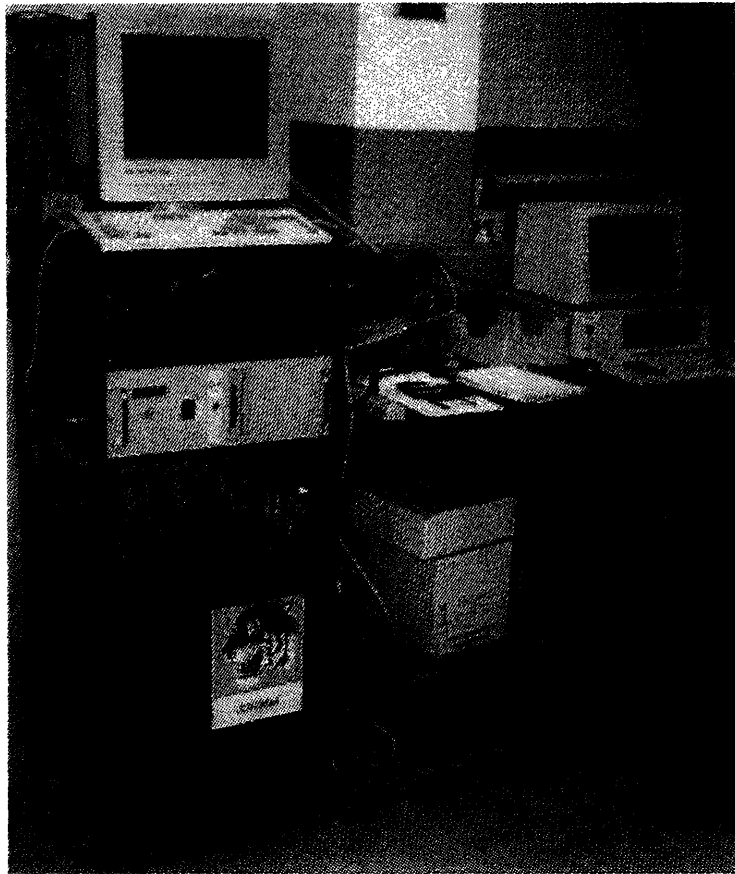


Figure I.5: The Final Setup

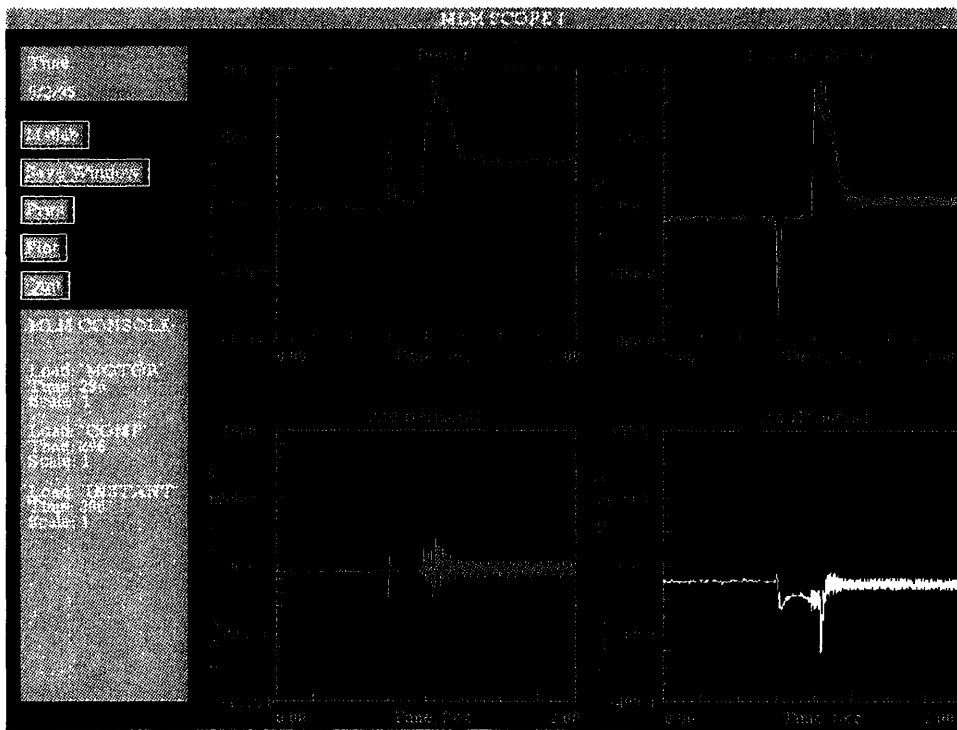


Figure I.6: MLM at Work: Computer, Small Motor, Instant

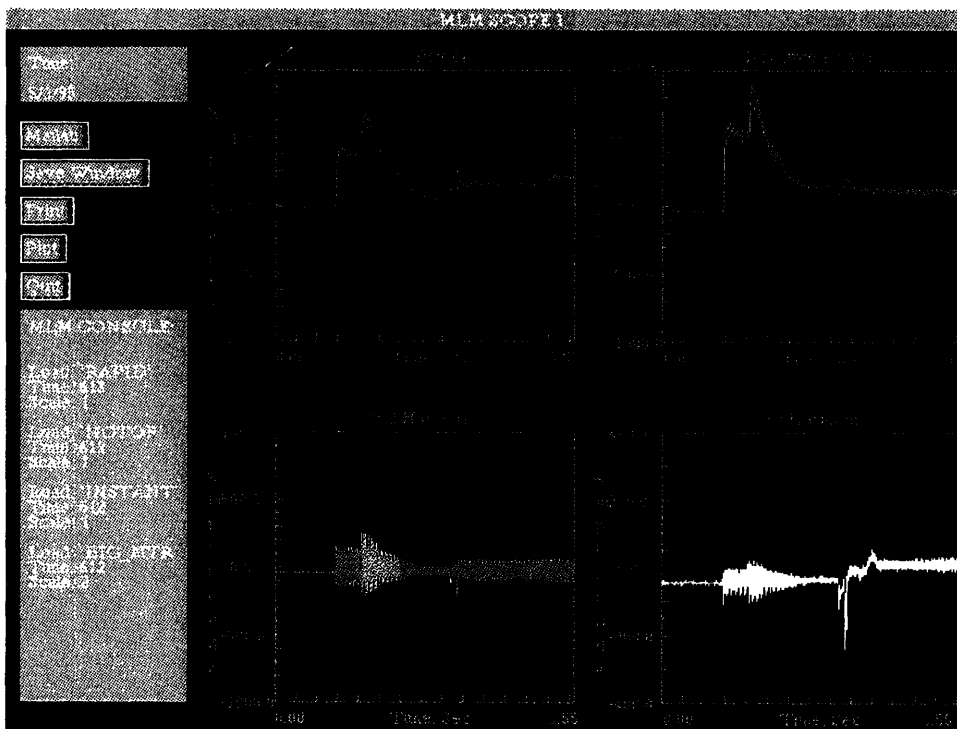


Figure I.7: MLM at Work: Big Motor. Small Motor. Rapid, Instant



Room 14-0551
77 Massachusetts Avenue
Cambridge, MA 02139
Ph: 617.253.5668 Fax: 617.253.1690
Email: docs@mit.edu
<http://libraries.mit.edu/docs>

DISCLAIMER OF QUALITY

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available. If you are dissatisfied with this product and find it unusable, please contact Document Services as soon as possible.

Thank you.

Some pages in the original document contain color pictures or graphics that will not scan or reproduce well.