National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

A USER-FRIENDLY SOFTWARE INTERFACE FOR THE LIQUID METAL CLEANLINESS ANALYZER (LiMCA)

by

Tudor Draganovici

A Thesis Submitted to the Faculty of Graduate Studies and Research

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering

Department of Mining and Metallurgical Engineering

McGill University

Montréal, Canada

ABSTRACT

The development of high quality metal products requires "clean" liquid metals as the base materials. For a large number of applications there is a need to quantify the cleanliness of the liquid metals in the sense that the number and size of inclusions have to be controlled to be below some acceptable limits. The demand for quality helped the development of measuring systems that can count the number and size distribution of inclusion. One of the devices, called LiMCA (Liquid Metal Cleanliness Analyzer), was developed at McGill University and has been successfully used in the aluminum industry for a number of years.

The LiMCA apparatus is based on the Electric Sensing Zone principle. By maintaining a constant current through a small orifice through which liquid metal passes, non-conductive particles passing through the orifice temporarily increase the electrical resistance of the orifice, therefore increasing the electric potential. The signal processing component of the LiMCA system detects the voltage transients, translates them into particle sizes, and counts them based on their sizes or stores the transients in certain time increments.

The current LiMCA system uses analog electronic components to implement the signal processing part and describes a transient only by its height or its time of occurrence. This implementation has limited the further development of the system for applications where the particle size distribution and particle occurrence must be counted concurrently.

Digital Signal Processing (DSP) technology has been successfully applied to upgrade the LiMCA system. With this technology, the DSP-based LiMCA system is able to describe each LiMCA transient by a group of seven parameters and to classify it into a certain category with the help of these parameters. Moreover, it counts the classified peaks based on their height (Pulse Height Analysis) and their time of occurrence (Multi-Channel Scan) concurrently for data acquisition.

A conceptually new software was designed to accommodate the DSP-based LiMCA and the Object Oriented Programming technique was used to develop the Graphical User Interface which constitutes the framework of the overall host interface.

RÉSUMÉ

Le développement de produits métalliques de haute qualité requière, à la base, des métaux liquides propres. Pour de plus en plus d'applications, la propreté du métal liquide doit être évaluée et le nombre et la taille des inclusions doivent être contrôlés en dela de valeurs acceptables. Ces besoins ont motivé le développement de techniques de mesure du nombre et de la taille des inclusions. L'appareil LiMCA (Liquid Metal Cleanliness Analyzer), développé à l'Université McGill et utilisé avec succès dans l'industrie de l'aluminium depuis quelques années, est une de ces méthodes.

Le fonctionnement du LiMCA est basé sur le principe de la Zone Électrique Sensible. Un courant électrique est maintenu à travers un orifice au bas d'un tube submergé dans un bain de métal liquide. Le métal liquide est aspiré à l'intérieur du tube et lorsqu'une inclusion non conductrice passe à travers l'orifice, elle augmente, pour un bref instant, la résistance électrique de l'orifice. Un système de traitement de signal détecte et mesure les transients, les convert en taille de particule, et les compte en fonction de leur taille ou, accumule les comptes par intervalle de temps.

Le système de traitement de signal du LiMCA actuel est constitué de modules d'électronique analogue. Il ne peut décrire les transients que par leur amplitude et par le temps auquel ils surviennent. Cette restriction freine le développement de l'appareillage LiMCA pour des applications où différents types de transients existent et doivent être classifié avant d'être traité.

Un nouveau système de traitement numérique des signaux a été conçu et mis en marche avec succès. Avec cette technologie, chaque transient est décrit par un groupe de sept paramètres. L'analyse de ces paramètres permet de classifier le transient. De plus, les distributions temporelles (Multi-Channel Scaling) et de taille des transients classifiés (Pulse Height Analysis) peuvent être obtenu simultanément.

La technique de programmation orientée objet a été utilisé pour développer l'interface usager et l'interface avec la carte de traitements numériques des signaux du nouveau logiciel LiMCA.

ACKNOWLEDGMENTS

This work was carried out under the supervision of Prof. G. Carayannis and Prof. R.I.L. Guthrie. The author is greatly indebted to them for their encouragement, academic and financial support during the course of my studies.

Special thanks to Prof. G. Carayannis for his valuable knowledge of computer technology and programming techniques that the author learned from him and applied into this work.

The author would also like to transmit his gratitude to Mr. F. Dallaire, McGill's Metals Processing Laboratory manager, for his cooperation in LiMCA experimenting and data processing and for his comments on the thesis.

Special thanks to Mr. X. Shi, a good friend and a valuable colleague of mine, for his daily collaboration and discussions during the progress of the present work.

Last but not least, I would like to thank my future wife, Daniela, for her constant and devoted support during the period that the author spent at McGill University.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1. General considerations

The presence of inclusions (i.e. foreign, undesirable particles) in metals can be detrimental to the properties of the final products. The continuously increasing demand for high quality, requires that metal cleanliness be monitored and described quantitatively. For some products (such as beverage cans, turbine blades, aerospace parts, etc.), both the number and the size distributions of inclusions present in the metal have to be controlled and kept below certain acceptable limits. Several inclusion measuring methods have been proposed [Pitcher and Young 69, Bauxman et al. 76, Siemensen 81, Levy 81, Bates and Hunter 81, Mansfield 82] but most of them are off-line techniques that require considerable amounts of labor and time.

A novel on-line method, known by the acronym LiMCA (Liquid Metal Cleanliness Analyzer) was developed at McGill University by researchers Doutre and Guthrie [Doutre 84]. This measuring technique has been successfully used for quality control in the aluminum industry by Alcan. It is worth mentioning that BOMEM Inc. has already started, with the approval of Alcan, producing and selling LiMCA machines for use in the aluminum industry. However the commercial LiMCA system doesn't have the flexibility to provide detailed information as required by researchers.

The fact that LiMCA is an on-line method gives it the potential to be used for the development of a process control system. At McGill significant amount of research has been carried out to verify the application of LiMCA to other metals and alloys, such as zinc, magnesium, copper, steel, etc. [Kuyucak 89, Kuyucak and Guthrie 89, Lee 91]. In addition to the uses of LiMCA to liquid-metal quality monitoring and control, there is a quite powerful trend to use LiMCA as a research tool in the studies of metallurgical processes. For example, in the study of ceramic foam filters for liquid aluminum, measurements were done to determine the concentration of inclusions upstream and downstream with LiMCA [Tian et al 92]. LiMCA was also used in the research on the kinetics of removal of Ca and Na from Al and Al-1wt%Mg by chlorination [Kulunk 92]. In the investigation of powder injection processes, an Aqueous Particle Sensor, which is a water version of the LiMCA system based on the same operating principle, was used [Yamanoglu 92].

Several researchers and industrial engineers have expressed strong expectations on the future applications of LiMCA in the studies of metallurgical processes and in particular in understanding and optimizing such processes. In general, a typical metallurgical process involves the interactions and reactions among liquid metal, solid inclusions and injection agents of different types, gas bubbles, and liquid inclusions. The metallurgists studying these processes would receive a great deal of help by knowing the size distributions and frequencies of occurrence of different types of inclusions at a certain location and a certain time.

The demand for such a tool for use in controlling and studying the metallurgical processes motivated our LiMCA research project. The final goal is to develop a system that can tell, to some extent, the operator what happened and what is happening inside the liquid metal in various processes. The work described in this thesis involves mainly the work related to the graphical user interface of LiMCA, but parts of the digital signal processing system that was developed in parallel will also be described. In the subsequent sections of this chapter, an introduction to the LiMCA system and its operational principle are presented.

1.2 LiMCA's principle of operation

The LiMCA principle of operation is the one used in the Coulter counter, an instrument originally developed to count blood cells suspended in a conductive fluid [Coulter 56]. By maintaining a constant current through a small orifice through which liquid metal passes, non-conductive particles passing through the orifice temporarily change (increase) the electrical resistance of the Electrical Sensing Zone (ESZ), which therefore result in transient changes in the electric potential.

1.2.1 Electric Sensing Zone principle

The LiMCA technique is based on the Electric Sensing Zone (ESZ) principle. Inside a liquid conducting media, an electrically insulating wall is installed to separate the media in two parts. A long cylindrical orifice is opened on the wall and through this orifice liquid metal can flow between the two parts. A constant DC voltage is applied across the orifice thus a steady electric field is formed in the conduction media and an overall voltage drop can be detected across the orifice. Because of the geometric confinement, the electric field is intensified inside the orifice making it very sensitive to the change of the homogeneity of the media inside the orifice. This change can be detected by measuring the voltage drop across the orifice, which we call the Electric Sensing Zone

(ESZ). A non-conductive particle suspended in the flow off the liquid through the orifice will inevitably increase the overall resistance of the ESZ and cause a voltage transient across it. The shape and magnitude of the transient are related to the characteristics of the particle as well as to other factors.

In Figure 1.1, a cross-section of a cylindrical orifice, together with a particle passing through it, is illustrated. The following assumptions are used:

1. The inclusions are spherical;
2. The inclusions are non-conductive;
3. The orifice is cylindrical with diameter D and length L (L>>D);
4. Only one inclusion passes through the orifice at a given time;
5. The current density within the ESZ is constant.

Based on these, a simplified relationship between the voltage change ΔV and the

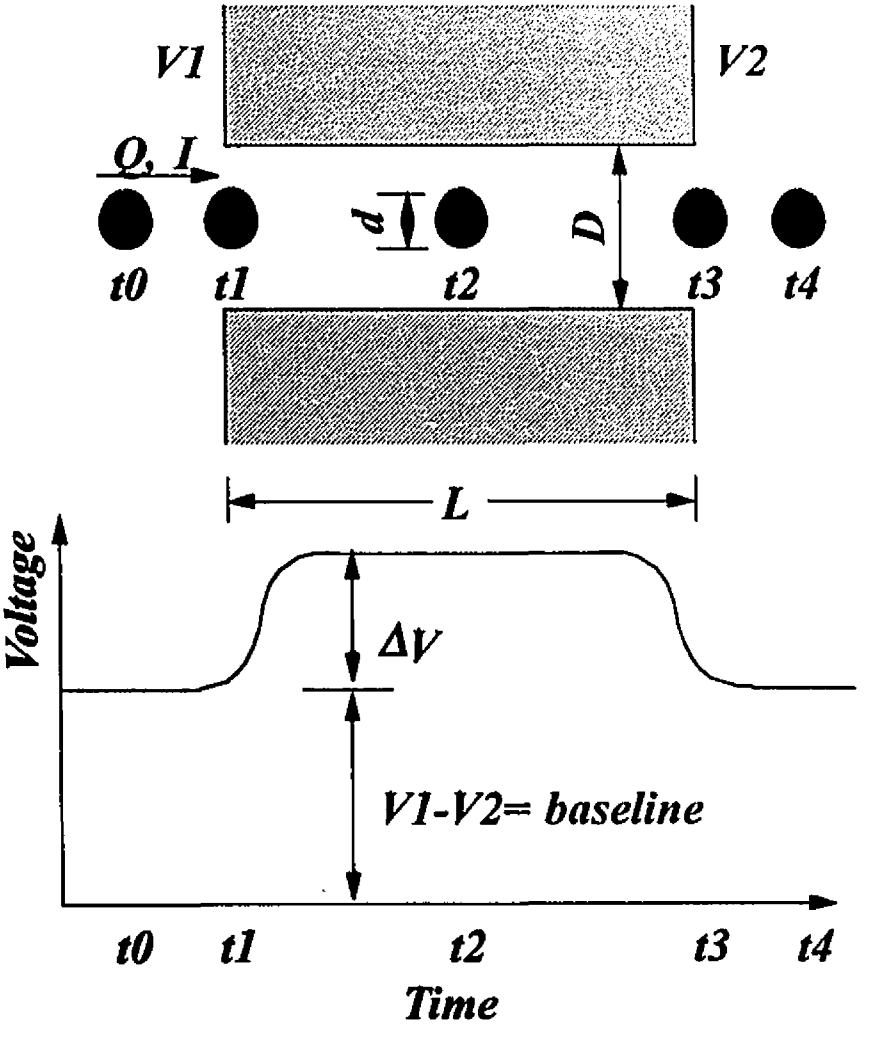

Figure 1.1 ESZ Principle

volume of the particle was given by DeBlois and Bean [DeBlois and Bean 70] and is used as a fundamental equation to predict the size of the particle from the voltage change (Equation 1.1).

$$\Delta R = I \frac{4 \rho d^3}{\pi D^4} \times f(d/D) \tag{1.1}$$

where I is the electric current;
ρ is the electrical resistivity of the melt (0.25 Ω·μm for liquid Aluminum);
d is the diameter of a spherical inclusion;
D is the diameter of a cylindrical orifice;
and f(d/D) can be expressed as:

$$f(d/D) = \frac{1}{1-0.8(d/D)^3} \tag{1.2}$$

1.2.2 LiMCA Sensor

The design of an ESZ based sensor for liquid aluminum is shown in Figure 1.2. It is designed to have an orifice of a certain shape and to capture the voltage change due to the passage of a particle through the ESZ. It consists of two electrodes and an electrically insulated vessel having a small orifice at its side wall, near its bottom. The tube is made of Kimax glass and the electrodes are made of steel. One electrode is positioned inside and the other outside the vessel, facing the orifice. Note that a number of variations of this design have been used in several experimental setup procedures designed for use both in aluminum and in other melts. During operation, the sensor is submerged in the melt, vacuum is applied inside the vessel to maintain a constant flow of metal through the orifice. A constant DC current (typically 60 A) passes between the two electrodes.

During the normal operation of the LiMCA, liquid metal is drawn into, or pumped out of, the vessel through the orifice. In general, inclusion particles suspended in the melt have conductivity much greater than that of the melt itself. When such a particle passes through the orifice, it displaces an equal volume of metal and thus causes a temporary change (increase) in the overall resistance of the ESZ. Since the electric current through the orifice is constant, this resistance change is detected as a voltage transient across the two electrodes. The shape and the height of the voltage transient are related to the shape, size, resistivity, velocity and trajectory of the particle within the ESZ, as well as to the metal flow across the orifice.

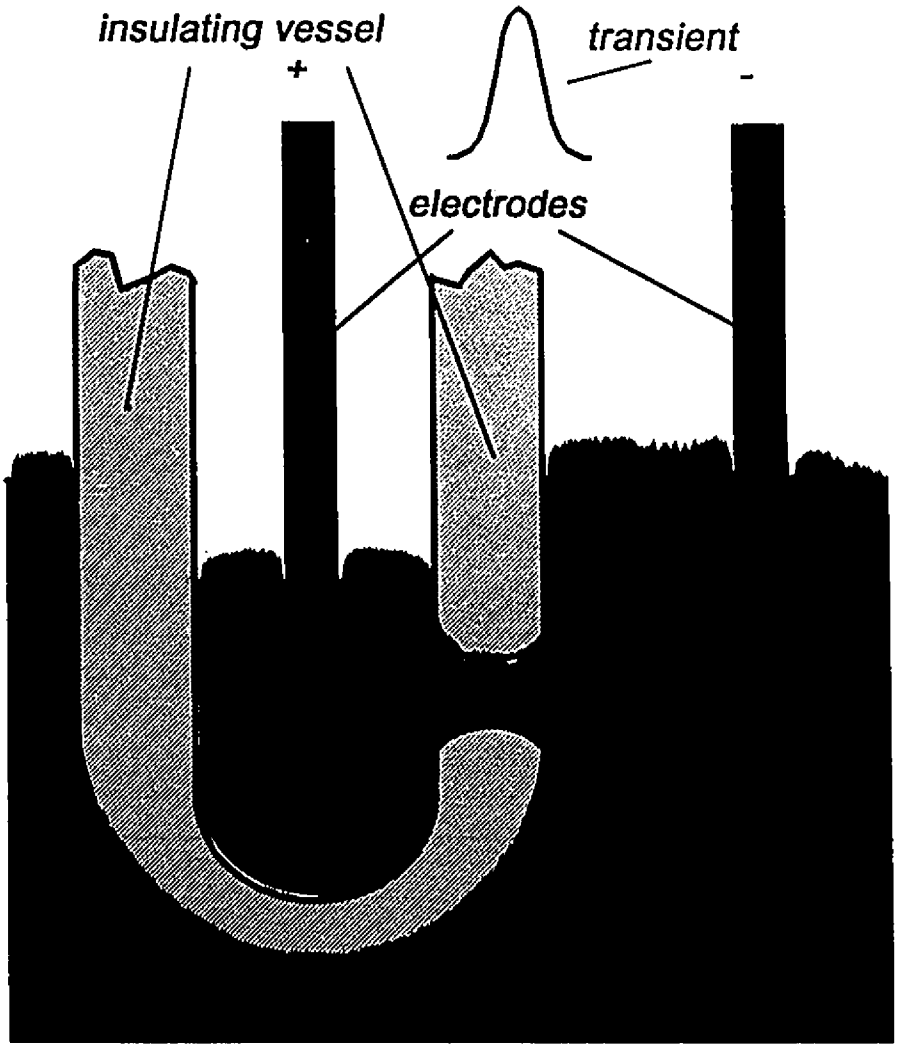

Figure 1.2 LiMCA Sensor

1.2.3 First Generation, Analog LiMCA System

The first generation LiMCA system, which was designed in the early 80's, is shown schematically in Figure 1.3. This system consists of four parts: the sensor, the power supply, the pressure and vacuum system, the signal conditioning system and an analog signal processing system. A battery is used as a power supply and provides the required constant current. A vacuum cylinder connected to a vacuum pump and a cylinder containing argon gas under pressure, are used to build the vacuum/pressure system.

The magnitude of the voltage transients that the system must detect is in the microvolt range. The transients are superimposed on a DC offset, which for aluminum is about 0.12 volts. This DC component corresponds to the constant voltage drop across

the orifice when no inclusion is present. The signal conditioning stage eliminates this DC offset, filters out high frequency noise, performs bandwidth reduction, and amplifies the signal to millivolt level for further processing. To increase the sensitivity to small pulses, the signal is passed through a logarithmic amplifier.

Further processing is carried out by an analog signal processing system, built from commercially available units. Here, a pulse sampler (model TN-1246, from Tracor Northern) is used to detect and measure the height of the transients and feed their magnitudes to a multi-channel analyzer (model TN-7200, from Tracor Northern). The latter has two modes of operation, one called PHA (Pulse Height Analysis) generating a size distribution and the other MCS (Multi-Channel Scaling) generating a time distribution

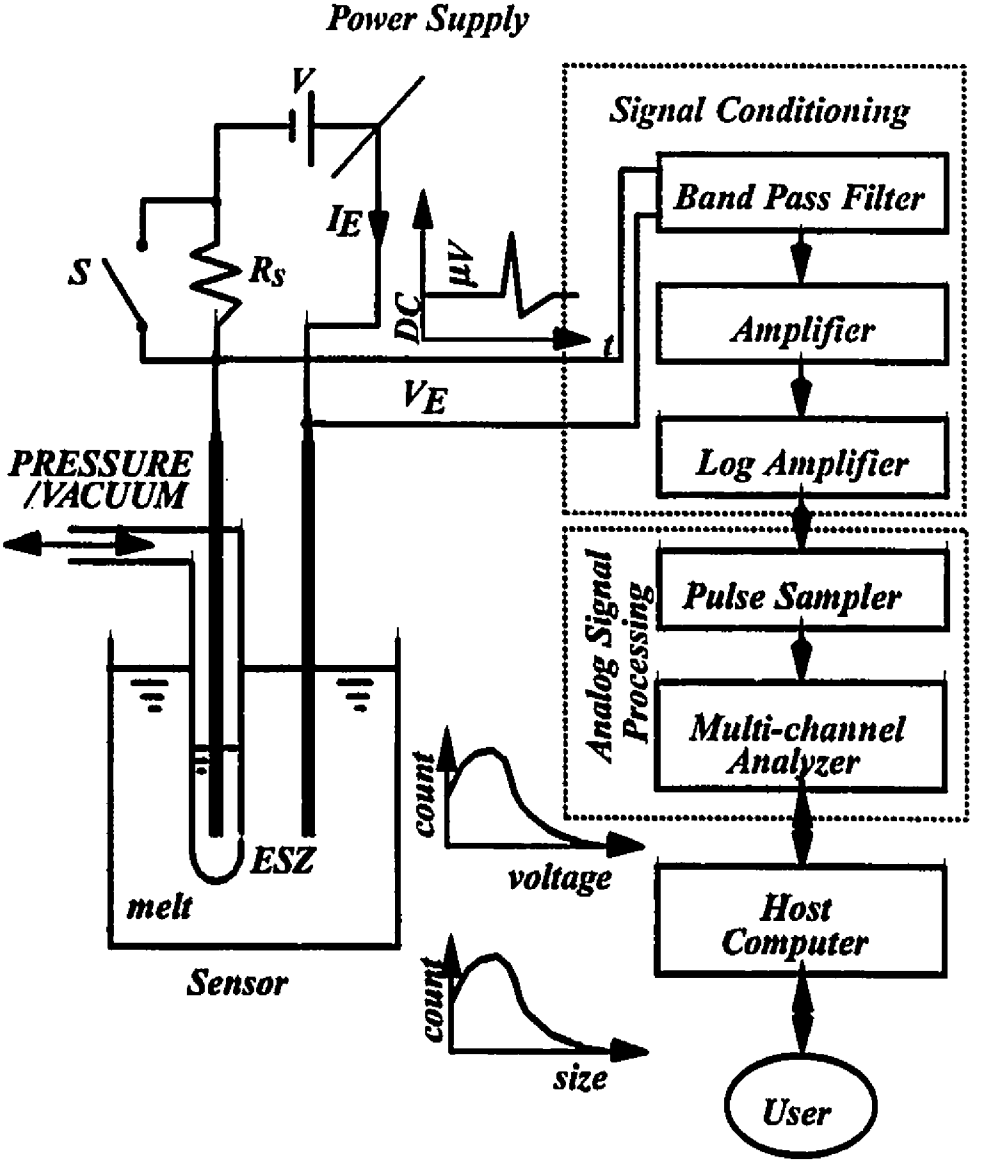

Figure 1.3 Analog LiMCA system

of the transients (Figure 1.4).

In the PHA mode, transients are classified according to their magnitudes. The voltage distribution is converted to an inclusion size distribution, using Equations 1.1 and 1.2. Together with the volume of metal sampled, it can be used to calculate measures directly related to metal cleanliness, such as the number of inclusions per kilogram of metal, the number of inclusions of given size ranges per kilogram of metal, the volume ratio of inclusions to metal, etc. One parameter called N_{20} is used extensively in the aluminum industry. It is defined as the number of inclusions whose diameter is larger than 20 μm per unit mass of liquid metal. N_{20} is the main output parameter of the industrial LiMCA system used to define metal cleanliness and is obtained by assuming that all transients detected are related to particles and that the fluid flow through the orifice is constant [Dallaire 90].

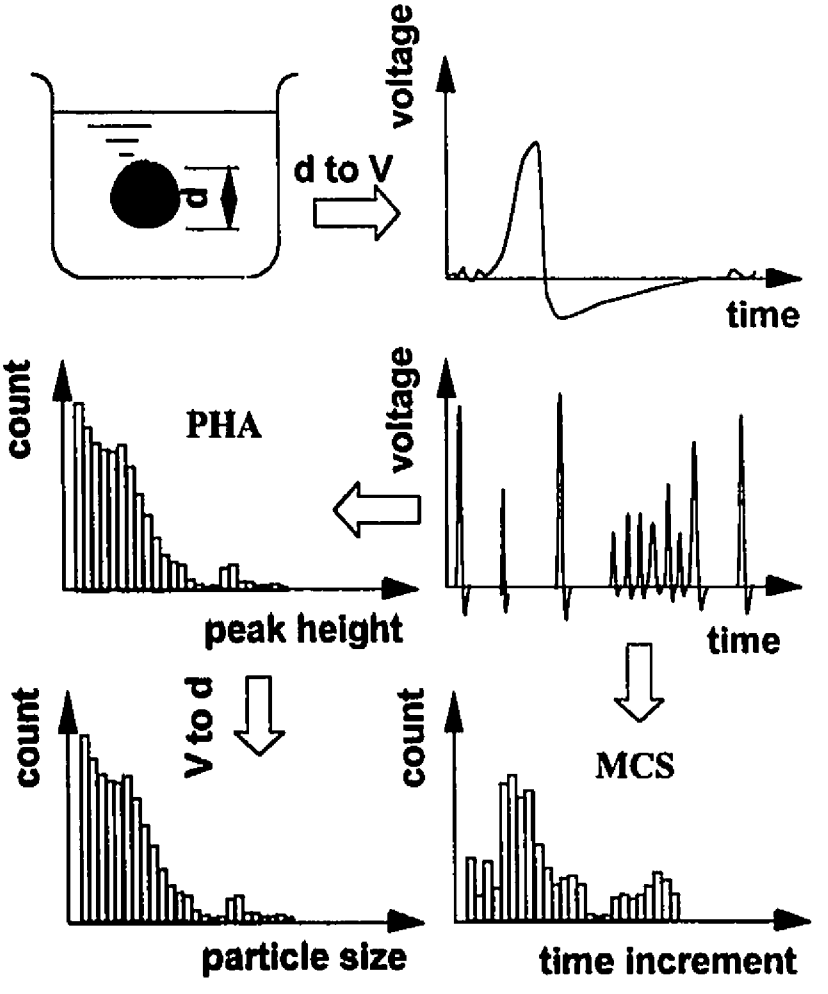

Figure 1.4 LiMCA data analysis procedures

In the second mode of operation (MCS), the multi-channel analyzer counts the transients that are detected within a certain time increment. The MCS mode gives the time distribution of inclusions at the location of the LiMCA sensor. Such information becomes more and more interesting to metallurgists for the study and control of certain metallurgical processes, such as, for example, the chlorination and the alloying process of aluminum [Kulunk 92]. Both operation modes are data analysis procedures and are illustrated in Figure 1.4.

A portable IBM-PC computer is used to acquire and process the data obtained by the Multi-Channel Analyzer (MCA). The MCA and the PC are connected through a serial RS-232 communication link. The data transfer allows for direct storage into DOS files in an ASCII format, limiting as much as possible human intervention. The time and date of file creation are always stored as file attributes. A BASIC program has been written to implement two specific functions:

1. Transfer data from the MCA to DOS files in an IBM-PC;
2. Data collection assistance to the operator of a LiMCA experiment.

Although this two-step procedure works well and was used successfully for a long period, one can notice that after the data acquisition process is finished, the ASCII file obtained has to be further processed (i.e. in a spreadsheet) in order to be able to represent the data in a useful way for the human operator. Another aspect that has to be taken into account is related to the new trend in the user interfaces community, that is offering to the human operator a graphical windows-type representation of the respective topic.

1.2.4 (Real) LiMCA Voltage Transients

In this section the different types of transients that are observed using the LiMCA system will be examined. A typical LiMCA signal, as measured in liquid aluminum, is presented in Figure 1.5.

In normal operation this type of signal appears the most frequent and is generated by the passage of an inclusion through the ESZ. This is the reason why we call such a signal a Normal Pulse (NP).

Other types of transients, having different characteristics than normal pulses, have been detected in different tests. In normal operating conditions, these new transients do not appear as often as the Normal Pulses. Such transients are shown in Figures 1.6 and 1.7 and are called Baseline Jump (BJ) and Negative Baseline Jump (NBJ) respectively [Dallaire 90].

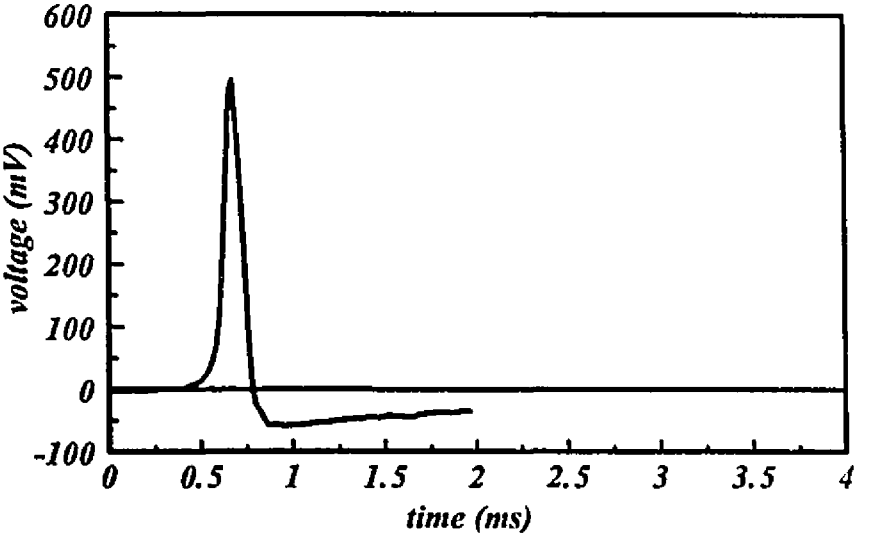

Figure 1.5 A typical Normal Pulse (NP)

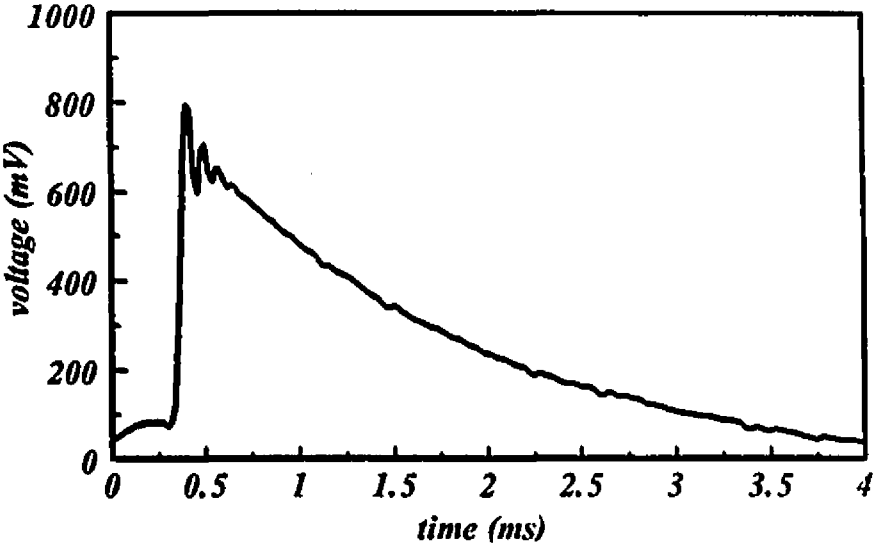

Figure 1.6 A typical Baseline Jump (BJ)

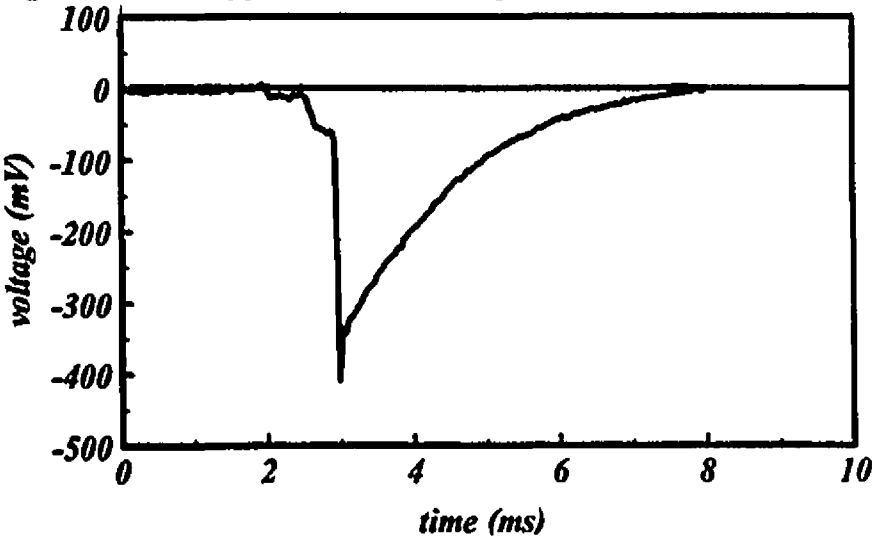

Figure 1.7 A typical Negative Baseline Jump (NBJ)

Their characteristics include a steep starting edge and an exponential trailing edge, restoring the baseline. The width (or time duration) of a BJ or a NBJ is usually several times greater than that of a NP, having the same magnitude. The most valid physical explanation for the appearance of such peaks is that they represent the response of the high pass filter (see Figure 1.3) to step changes in the resistance of the ESZ. Several physical phenomena at the ESZ can result in such a step change in resistance: partial obstruction of the orifice, expansion or reduction of the orifice. Also, a long cylindrical inclusion passing through the orifice in its longitudinal direction would give rise to this type of transient.

On rare occasions, when several particles pass through the orifice at the same time,

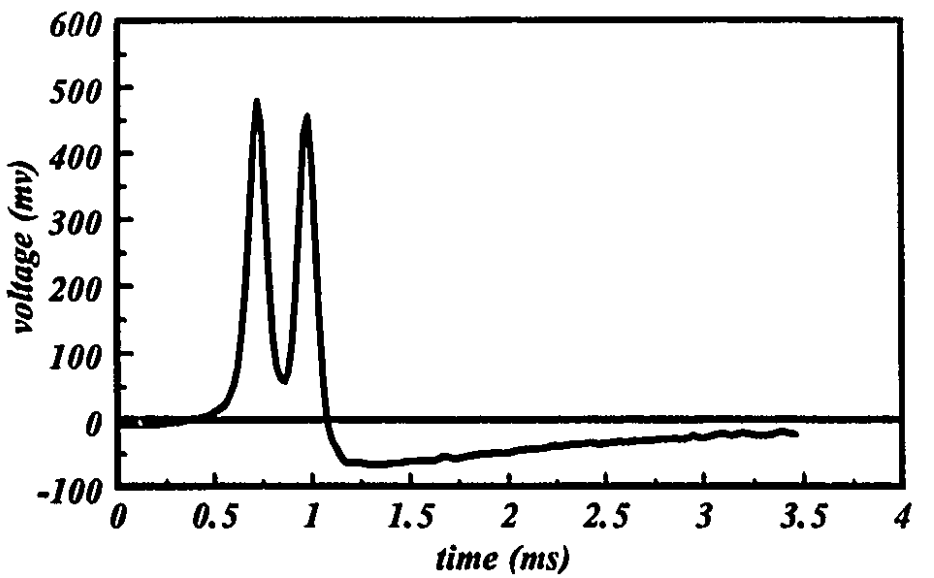

Figure 1.8 A Multiple Pulse (MP)

transients having more than one peak are detected. Such a signal, called Multiple Pulse (MP), is shown in Figure 1.8. Here, two inclusions were present in the ESZ at the same time.

In addition to the signal types mentioned above, two more have been identified. They are known as the Baseline Fluctuation (BF) and the Negative Baseline Fluctuation (NBF). The exact time domain shapes of these two types of signals vary. The appearance of such transients indicates oscillations of the baseline (i.e. the magnitude of the DC component) and consequently signals improper system operation.

1.3 Motivation and scope of present work

In the first generation LiMCA system, all transients, having magnitudes higher than a certain noise threshold are detected, their heights are measured and converted into the sizes of the corresponding inclusion particles. However, from our previous discussion it is evident that only NP type transients correspond to particles. BJ type transients may be related to particles but in most cases, they indicate other ESZ phenomena, such as reduced metal flow, partial blockage of the orifice, orifice size change, etc. Therefore it is important to develop a LiMCA system that can discriminate and classify the different types of transients. The first objective is to upgrade the first generation LiMCA so that different types of transients can be differentiated and processed differently.

The first generation LiMCA system (Figure 1.3) uses general purpose analog signal processing equipment (e.g. pulse sampler, multi-channel analyzer, oscilloscope). It detects only positive peaks and uses only one peak description parameter, the peak height. This hardware architecture does not provide the flexibility required to achieve the differentiation and classification mentioned before. As a result, the design of a software based LiMCA system using DSP (Digital Signal Processing) technology was considered.

The first stage of development is to use DSP technology to elaborate a second generation LiMCA system, which is functionally equivalent to the first generation one. This stage is necessary in order to ensure compatibility between the two systems and also to facilitate the validation of the new one. The second stage involves the development of the required code so that the new system can automatically identify the different types of transients. The final goal is to integrate into the system a higher level of reasoning that can process the classified transients, and using knowledge about the metallurgical process, arrange each inclusion into one of a number of expected classes (based on composition, shape, state, etc.), and to develop a sensor that can be used, not only for quality, but also for process control.

It is clear from the previous paragraph that the DSP technology cannot do the whole job by itself. The overall task of the system includes real-time signal processing and high-level signal analysis. The real-time signal processing requires a high computational speed and for the high-level signal analysis advanced algorithms must be developed and implemented. Because of the different requirements of the two computational aspects, a multi-processor system is an ideal hardware environment. A DSP processor is the processor "responsible" for real-time signal processing and an IBM compatible personal computer acts as a host providing an interface for the human operator, communicating with the DSP processor, and executing the signal analysis. Therefore, two levels of

software must be developed at the same time: the DSP software running at the DSP level and the host interface software.

The development of the DSP-based LiMCA can be divided into several tasks. In terms of signal processing, we identify five tasks: the peak sampling process, the peak description process, the peak classification process, the extraction of size, shape and volume information of inclusion particles, and last, the development of an intelligent system which uses the information extracted from the NPs and the frequency of occurrence of the other types of transients together with the knowledge about the specific metallurgical processes and makes intelligent suggestions to the operator. Figure 1.9

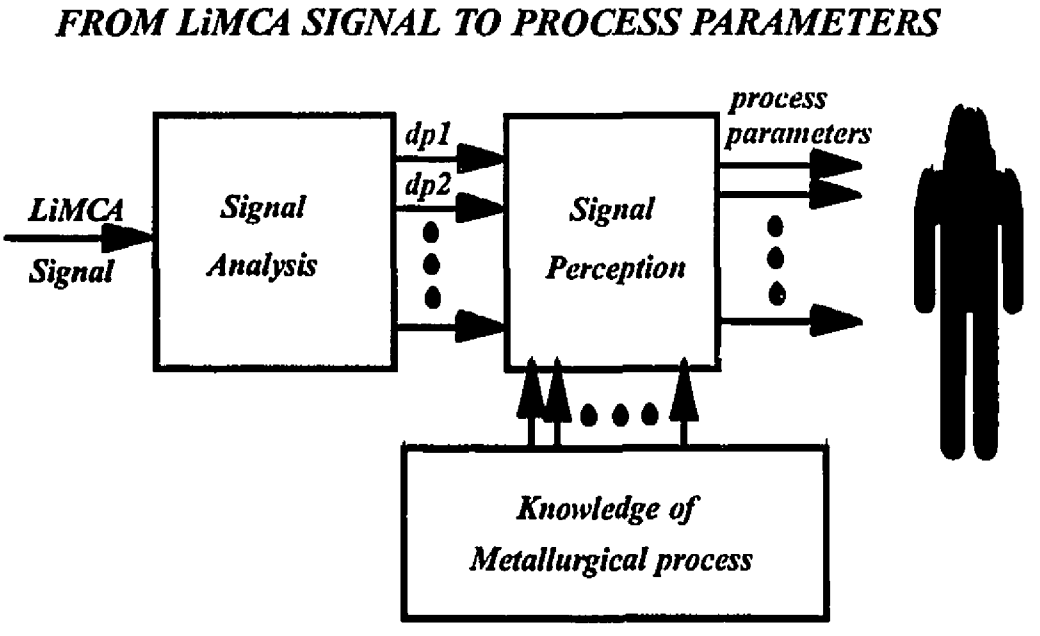

Figure 1.9 From LiMCA signals to process parameters

shows this process which is conceptually divided into the signal analysis part, that generates a description of the detected transients and labels them into associated types, and the signal perception part, which identifies the detected particles.

The tasks for the host interface software cannot be easily defined because of the complicated nature of the desired objective. One must take into account several facts: the signal processing algorithm is not immediately available, the performance of the DSP interface has to be tested, another types of algorithm and parameters have to be available for research purposes. Regarding the LiMCA operation, there are many parameters to handle and different configurations to be tested. To start the development of this

software, one has to consider a good software frame that can be easily reconfigured, upgraded, and easy to operate.

A graphical user interface based on the Object Oriented Programming technique should first be developed as the framework of the overall host interface. This should accommodate all the computational tasks and the host-DSP operation.

This thesis is concerned with the first stages of development of the new DSP-based LiMCA system focusing on the user interface part of the software. The user interface is designed to be a friendly and useful tool for the LiMCA users especially in a research environment.

In the subsequent chapters, the hardware and software of the DSP-based LiMCA will be discussed. Following this, the graphical user interface will be presented and finally, the conclusions of my research work and suggestions for future developments are included.

Chapter 2: GRAPHICAL USER INTERFACE DESIGN CONSIDERATIONS

Before explaining the Graphical User Interface, the complete program including the DSP process, which provides the real-time data, and the host analysis process, which represents the data at the host level, are presented. The host tasks are implemented as part of the Graphical User Interface and in order to understand the design and implementation, the DSP hardware and software need to be discussed.

In this chapter, a brief introduction to Digital Signal Processing (DSP) and a comparison between the digital and analog signal processing implementations are given, followed by a description of the new DSP-based LiMCA system (Section 2.2). In the second part of this chapter (Section 2.4), the principles of Graphical User Interfaces (GUI) are discussed. Their application in this particular LiMCA implementation are presented in the next chapter.

2.1 Digital versus Analog Signal Processing Implementation

Signal processing is concerned with the representation, transformation, and manipulation of signals and of the information they contain. For example, we may wish to separate two or more signals that have somehow been superimposed or to enhance some component or parameter of a signal model.

Digital signal processors (often called DSPs) are microprocessors with specialized architectures and instruction sets, designed to perform well in digital signal processing-intensive applications. There are more applications for DSP today than ever before, and this trend is projected to continue. DSP chips are replacing many analog solutions with improvements in cost, performance, and reliability. Designs with fixed features and preset parameters are being redesigned with DSP chips for flexibility and future upgradeability. These advantages do not come for free; the design cycle for DSP is a new and unfamiliar process for most engineers. Additionally, DSP algorithms bear no resemblance to the analog circuits they replace. After the algorithm is fully defined and fine-tuned, it must be implemented in DSP assembly-language code to run quickly. Once the DSP code is written, in order to test it, one has to run the code on the specific hardware to correct all the problems. To make this process easier, a number of DSP development boards were developed. The advantage of using such boards plugged into a computer is obvious: no custom hardware needs to be designed yet, thus saving development cost. The typical development board has the general-purpose circuitry on-board to cover most DSP

designs, such as static RAM, analog to digital and digital to analog converters, and other types of circuits. To assist in DSP code development, debugger monitor software is included with many boards.

The differences between the two implementations are illustrated in Figure 2.1. In the analog implementation (Figure 2.1 (a)), the original signal is processed by dedicated analog circuits or systems built from commercially available electronic devices. Then the output of the analog signal processing module can be displayed on a monitor and saved (e.g. on the hard-disk of a computer).

In the DSP implementation, the original signal is first digitized by an analog to digital converter (ADC). Then the digital signal processing software is executed at the DSP-board level. As mentioned before, this board is controlled by programs running on the DSP. These programs are developed and updated in agreement with the signal processing tasks. The DSP board is typically supervised by a host computer that receives the results of the digital signal processing through a communication link.

The major advantage of digital over analog signal processing is flexibility. Another point is the fact that the DSP-based implementation is software-based. Thus, it is easier to reconfigure the system to adapt to new conditions and parameters. Complex algorithms can be integrated into the DSP programs to improve the overall performance of signal

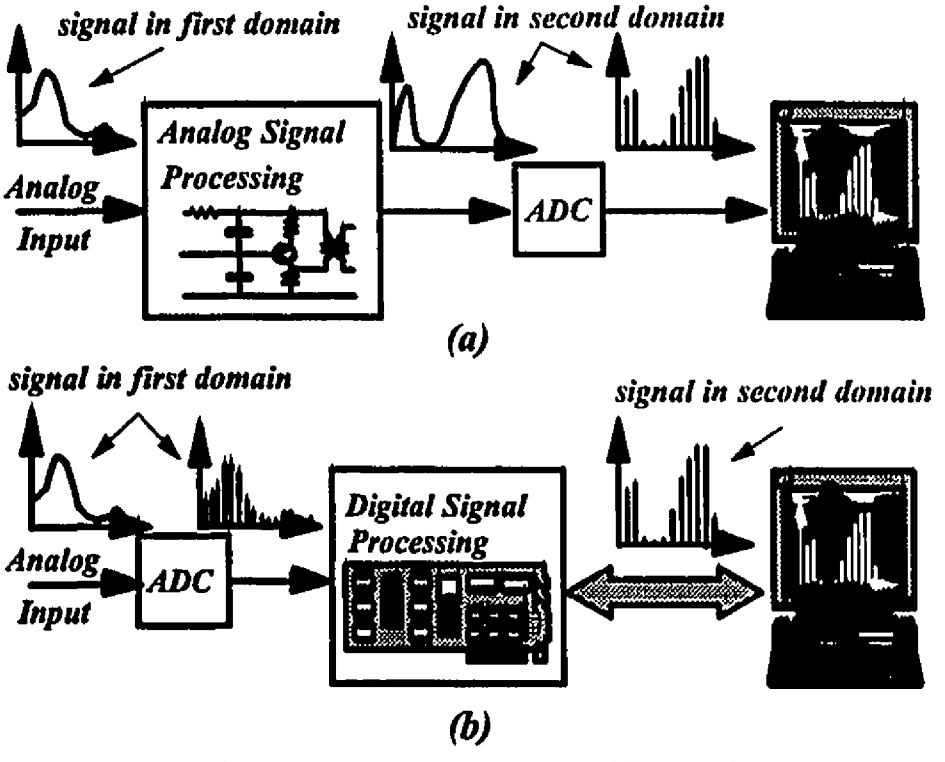

Figure 2.1 (a) Analog vs. (b) Digital Signal Processing

processing. Such improvements are difficult to accomplish with a dedicated analog signal processing circuit.

In addition to the advantages related to flexibility and cost, the most important aspect in the LiMCA application is related to the parameters that describe a peak. The analog signal processing system of the first generation LiMCA (Figure 1.3) describes LiMCA peaks with only one parameter in either PHA mode or MCS mode (see Section 1.2.3). In the PHA mode, the peaks are measured by the magnitude and height of the peaks is used by the PHA module to find the associated PHA channel. In the MCS mode, the peaks are labeled with time and accumulated within certain time intervals represented by MCS channels. As mentioned earlier in Section 1.2.4, different types of peaks can be observed in LiMCA operations. Each of them is a result of different ESZ phenomena. For better understanding of these phenomena and for a correct interpretation of the LiMCA signal, the types of peaks should first be classified into different categories. Furthermore, from Section 1.2.2, the shape of a peak carries the shape information of the inclusion. Thus, the description of a peak should also include the shape parameters. The real-time classification method was not available and is one of the major parts of this research work. The multi-parameter peak description and uncertain method of peak classification causes the complexity of the signal processing of the LiMCA system.

To summarize, it is not practical to design and implement a signal analysis system using an analog signal processing method and a DSP-based approach is the more attractive option.

2.2 DSP System General Outlook

The block diagram of the DSP-based LiMCA system is shown in Figure 2.2. By comparing to the first generation LiMCA system shown in Figure 1.3, one can notice that in the analog components such as the log amplifier, pulse sampler and multi-channel analyzer, are replaced with a digital signal processor.

The DSP processor is plugged into a PC host computer that is used to interface down to the DSP processor and up to the operator through a recently developed graphical user interface.

2.2.1 DSP Specifications for the LiMCA system

Most DSP processors share some basic common features, designed to support high-performance, repetitive, numerically-intensive tasks. The most often cited among those features is the ability to perform a multiply-and-accumulate operation (often called a

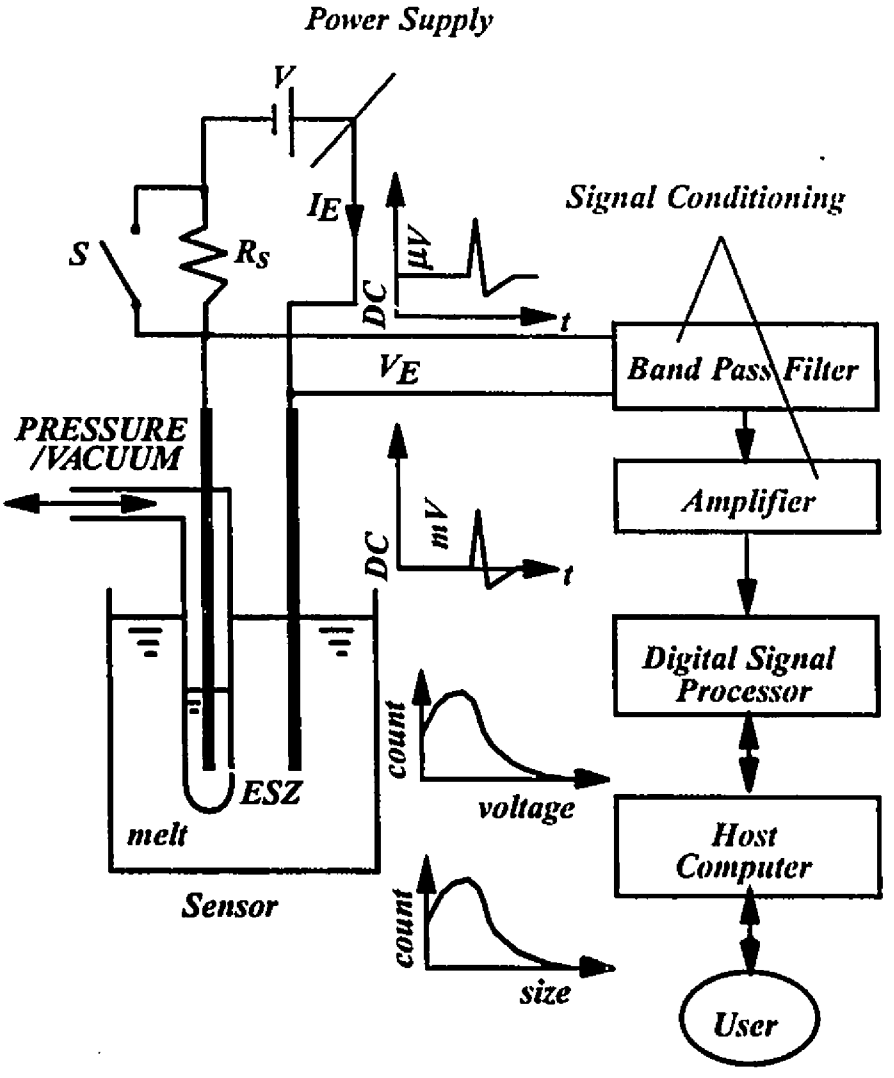

Figure 2.2 DSP-based LiMCA System

"MAC") in a single instruction cycle. The multiply-and-accumulate operation is useful in algorithms that involve computing a dot-product, such as digital filters.

A second feature shared by DSP processors is the ability to complete several accesses to memory in a single instruction cycle. This allows the processor to fetch an instruction while simultaneously fetching operands for the previous instruction, and/or storing the result of the previous instruction to memory. In general, such single-cycle multiple memory accesses are subject to many restrictions. Typically, all but one of the memory locations accessed must reside on-chip, and multiple memory accesses can only take place with certain instructions.

To provide simultaneous access to multiple memory locations, DSP processors provide multiple on-chip buses, multi-ported on-chip memories, and in some cases

multiple independent memory spaces. To allow arithmetic processing to proceed at the maximum speed possible, DSP processors incorporate a dedicated address generation unit. Once the appropriate addressing registers have been configured, the address generation unit operates in the background, forming the addresses required for operand accesses in parallel with the execution of arithmetic instructions.

In the case of the LiMCA system, some analysis of the signals has been done in order to establish the key parameters to be taken into account when choosing the DSP processor. Based on the signal processing tasks discussed in Section 1.2.3 and on the frequency analysis of the signals (see [Shi 94]), the most important DSP parameters were determined:

• Number of bits for analog-to-digital conversion

The number of bits used to present an analog value after an analog-to-digital conversion provides the resolution of the digital presentation of the analog signal. Presently 16 bit analog-to-digital converters (ADC) are common and appropriate for most applications. The absolute quantization error is less or equal to $\mathbf{X_m / 2^B}$, where **Xm** is the full analog input range and **B** is the bit length of the analog-to-digital converter [Oppenheim and Schafer 89]. The relative quantization error is thus within $\mathbf{1/2^B}$. For 16-bit ADC, the relative quantization error is 0.0000153 at maximum and produces a Signal-to-Noise ratio of 96 dB, which is much higher than that of the LiMCA signal of 36 dB [Shi 94].

• Sampling frequency

The sampling frequency of the ADC is determined according to the frequency components of the analog signal. In the case of a Normal Pulse, the frequency range is from 0 to 14 kHz. According to the Nyquist Sampling Theorem [Oppenheim and Schafer 89], the sampling frequency must exceed two times the maximum frequency of the signal. Consequently, the minimum sampling frequency is 28 kHz. Considering the other LiMCA peaks (i.e. baseline jumps and normal pulses), the frequency range is from 0 kHz to approximately 18 kHz. Therefore, the minimum sampling frequency to avoid aliasing for LiMCA peaks is 36 kHz. To ensure the accuracy of the signal processing, some over-sampling is also desired. As a result, a sampling frequency of 50 kHz is used.

• Input channels

As discussed in Section 1.3, measurements of two locations need to be compared in real-time. To cope with this type of application, the new LiMCA signal processing system must be designed to have two processing units being able to work simultaneously. Therefore, only DSP chips with two input channels are considered.

• Computational speed

The speed of the system is evaluated by the clock frequency of the DSP processor to be used. The required computational speed is considered according to the overall real-time signal processing task and the parameters discussed above. As discussed in Section 1.4, the overall task for LiMCA signal processing includes peak sampling, peak description, and peak classification processes. Also, referring to Figure 2.1, one can notice that a generic process, the analog-to-digital conversion, is always needed for digital signal processing. Considering a sampling frequency of 50 kHz, there are only 20 μ seconds available for these processes. Therefore the DSP board must be fast enough to guarantee that the processes can be finished within this time in order to process LiMCA signals in real-time.

The clock frequency can be estimated by using the maximum number of clock cycles for the ADC process multiplied by the sampling frequency plus the maximum number of clock cycles needed for the rest of the data processing tasks multiplied by the peak frequency.

The peak frequency in the worst case is 2000 peaks per second. This occurs when all detected peaks are Normal Pulses, closely following one another. The result of the calculation shows that the computational speed of the DSP processor must be faster than 12 MIPS (Million Instructions Per Second) and the clock frequency of the processor must exceed 24 MHz [Shi 94].

In summary, the essential requirements for the DSP board include two input ADC channels with 16 bit resolution, up to 50 kHz sampling frequency, and a system clock faster than 24 MHz.

2.2.2 DSP-board Hardware Configuration

Considering the above specifications, a DSP-56 coprocessor board for IBM PC type computers from Ariel Corporation, was chosen as the real-time DSP engine. Figure 2.3 shows the hardware configuration of the system.

A 50 MHz 80486-based computer is used as a host. The DSP-56 is based on the Motorola 56001 processor that runs at 27 MHz with an instruction cycle equal to 74.1 nanoseconds. The memory of the processor is organized in three 64 Kx24-bit sections, each with separate address and data buses. One section is used for program memory and the other two for data (X and Y data memory).

The DSP-board has two 16-bit ADCs (Analog-to-Digital Converter) and two 16-bit DACs (Digital-to-Analog Converter) channels. The sampling rate of the ADC is selectable from 16 choices ranging from 2 kHz to 100 kHz in the so-called 16-bit stereo mode. In this mode, signals from two LiMCA sensors can be acquired and processed concurrently. A high speed mono ADC mode with sampling rates up to 400 kHz is also available. An on-board SCSI (Small Computer System Interface) bus is also available. This can be used to connect a hard-disk so that data can be saved in real-time and then used off-line.

The DSP-56 also has one input/output bit that we use to interrupt the host computer whenever the real-time DSP process needs the attention of the host. The analog signal from the signal conditioning stage is connected to the ADC for real-time processing. A digital tape recorder (Model RD-101T, from TEAC) can also be used to record the

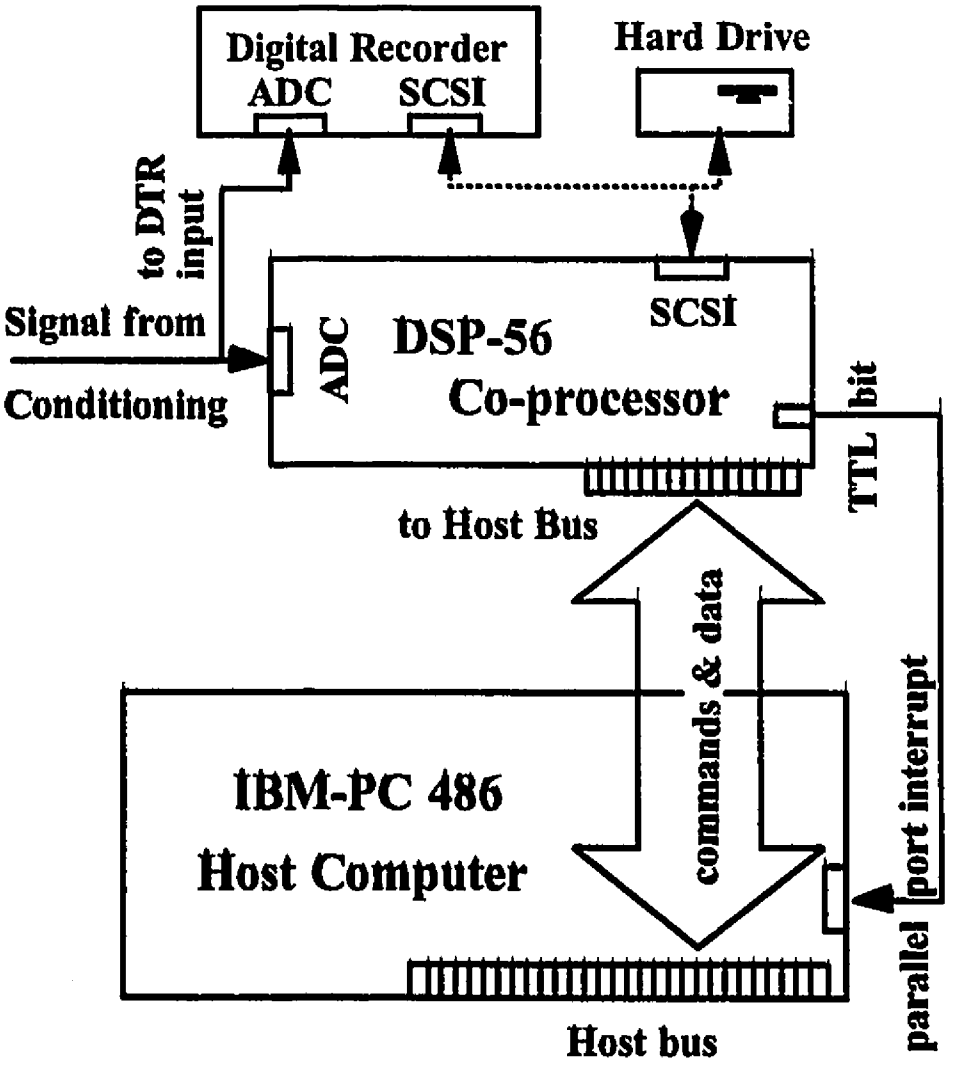

Figure 2.3 Digital Signal Processing Hardware

signals for later off-line analysis.

2.3 DSP Processes and Output

The software for the DSP LiMCA has been developed based on the hardware described in the previous chapter. It includes the real-time DSP software, a host PC-DSP-

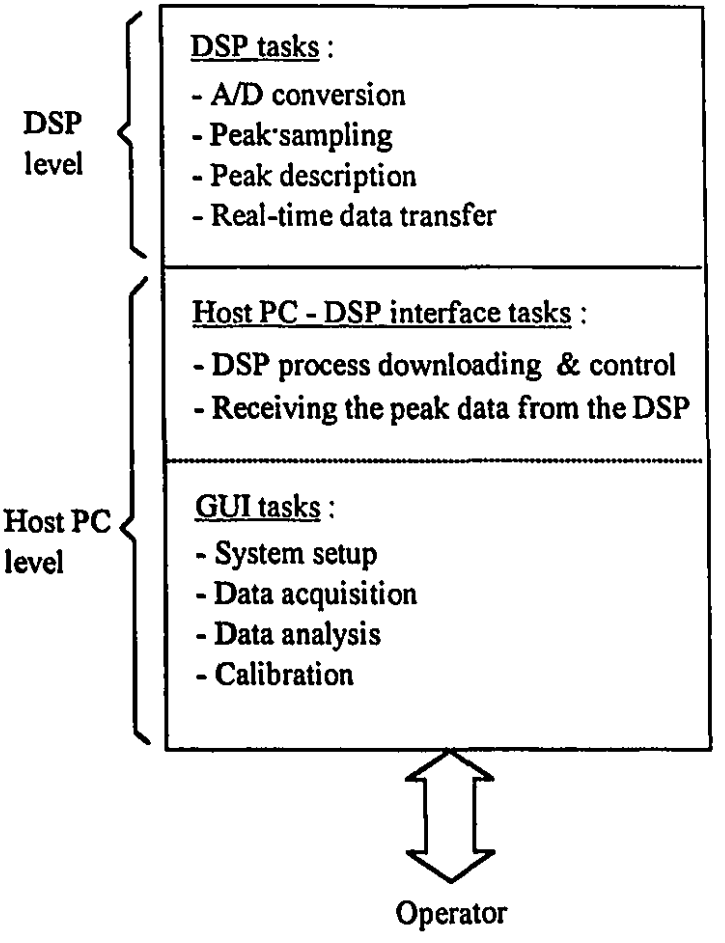

Figure 2.4 DSP & host PC tasks

based interface and a Graphical User Interface (GUI). The real-time signal processing is performed by the DSP software, which has been implemented using the Motorola DSP 56000 assembly language running in the DSP-56 coprocessor. The host PC-DSP interface provides the communication link between the host PC and the DSP board and downloads the DSP code and the configuration parameters from the host to the DSP. During the

execution of the DSP program, the real-time data are uploaded from the DSP board to the host PC through the PC-DSP interface.

The Graphical User Interface enables the LiMCA operators to control the instrument and provides a friendly environment with well-organized windows containing input fields, dialog boxes and graphical displays. At the same time, it performs all the host level computation tasks and controls the DSP processes through the host-DSP interface. The tasks at both the DSP and the host level are listed in Figure 2.4.

The host PC-DSP interface and the GUI were written in the C++ language and compiled with the Borland 3.0 C++ compiler. Two commercial software packages, object-Menu from Island Systems and MetaWindow from Metagraphics Software Corporation, were also used to implement the two interfaces.

A description of the DSP real-time software and of the host PC-DSP interface will be given in the next sections. Since this is not within the scope of the work described in this thesis, details concerning this part of the software are not presented here and can be found in [Shi 94]. The GUI is discussed in the next chapter.

2.3.1 DSP Real-time Software

The software developed at the DSP level is organized as a number of independent tasks. Each task is designed as a filter, reading data from an input buffer and writing new data into an output buffer. Figure 2.5 shows these tasks together with the corresponding data flow paths. A small executive program was developed to manage their execution. During system initialization, the executive receives a number of parameters from the host and starts the execution of the different DSP tasks.

Note that Figure 2.5 shows a one-channel system. The analog signal from the signal conditioning stage is digitized by the ADC. An Interrupt Service Routine (ISR) is invoked which reads the output of the ADC and writes the data into a circular buffer. The circular buffer is processed by the Peak Sampling Process that detects the presence of peaks and transfers peak data to the "sampled peak buffer". A digital filter can be invoked before the peak sampling process to eliminate "known" noise, for example from an induction furnace near by. The "sampled peak buffer" is processed by the Peak Description Process which stores its output into the "peak buffer".

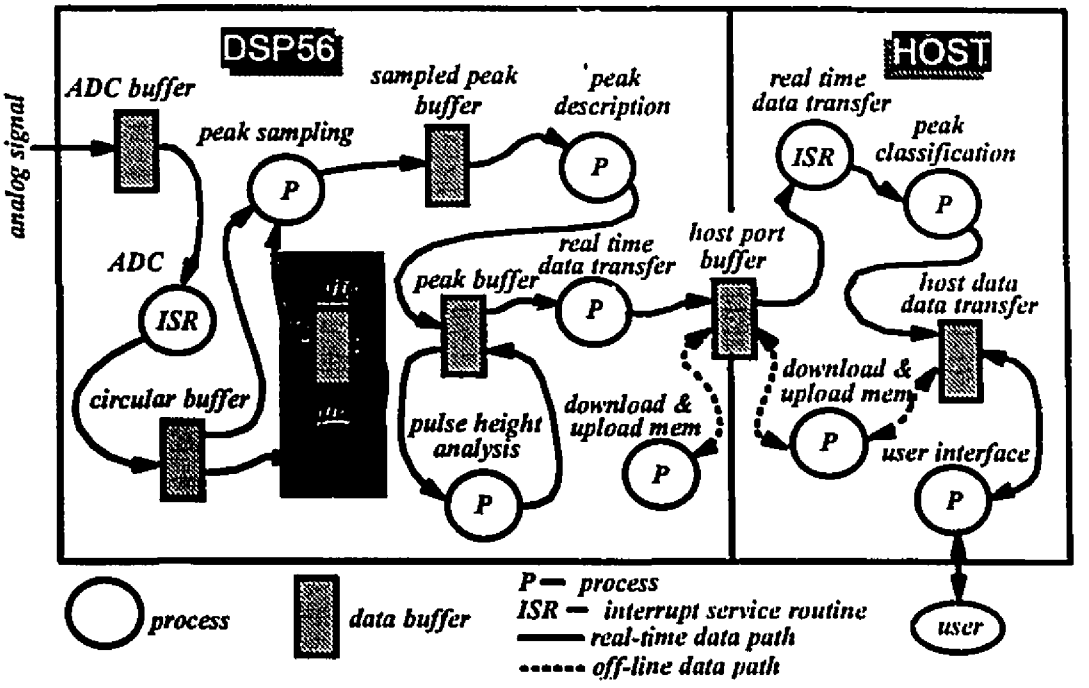

Figure 2.5 DSP real-time software structure

The Pulse Height Analysis Process computes and modifies this information. The results are transferred to the host PC through the 24-bit Host Port. At the host level, an ISR is invoked to read the data from the Host Port and transfer them to the Peak Classification Task.

These processes were first distributed between the DSP-56 coprocessor and the host computer, and then coded separately. The scope of this static task allocation is to take full advantage of the pipeline architecture of the DSP and to maximize its utilization. The pipeline architecture refers to instruction pipelining which allows overlapping of instruction execution so that the fetch-decode-execute cycles of a given instruction occur concurrently with the fetch-decode-execute cycles of the next and previous instructions. Specifically, while an instruction is executed, the next instruction to be executed is decoded, and the instruction to follow the one being decoded is fetched from the program memory. Pipelining is normally transparent to the user. Time is "wasted" when the DSP coprocessor board communicates with the host. Due to the above explained pipeline architecture of the Motorola DSP 56001 processor and the DSP-56 board, two types of operations are the most time consuming: control transfer instructions and instructions that perform data transfers between the DSP-56 and the host PC. The majority of the data transferred are peak description parameters. Therefore, the total number of parameters used for peak description has to be minimized in order to minimize the data transferred.

The peak sampling and the peak description tasks perform most of the data reduction and must therefore be programmed at the DSP level.

In the initial prototype of the DSP-based LiMCA system, the peak classification process was implemented at the host computer level. This decision was influenced by the fact that the system was being developed for a research environment and will be used in different melts and under different working conditions. By programming the peak classification at the host PC level, it is easier to adapt the software to handle different situations.

2.3.2 Peak Parameters in the DSP-based LiMCA

From Figure 2.5, one can observe that the peak sampling process gets data from the "circular buffer", detects the presence of a transient and writes its output to the "sampled peak buffer". In general, a LiMCA peak can be positive, negative or a combination of the two (e.g. a Normal Pulse) as discussed in Section 1.2.4. The peaks can have different widths and different starting and trailing slopes.

This process compares the incoming data with two noise thresholds, filtering out small baseline oscillations. The thresholds depend on the noise levels at the site of the measurements and on the size of the smallest peak that must be detected. If a data point is higher than the high noise threshold or lower than the low noise threshold (Figure 2.6), the presence of a peak is detected and the Peak Description Process is activated. Using this algorithm, a normal pulse can sometimes be detected as two peaks, a positive peak and a negative undershoot. This happens when the magnitude of the positive peak is so big that its undershoot falls under the lower noise threshold. The undershoot can be easily distinguished because it follows closely a Normal Pulse.

The Peak Description Process analyzes data from the "sampled peak buffer" and generates a six parameter description of each pulse. These include three shape and three time parameters (Figure 2.6). The shape parameters are the *peak height*, the *peak starting slope* and the *peak ending slope*, and the time parameters are the *peak start time*, the *peak end time* and the *time at the peak's maximum* or *minimum point*, for positive and negative peaks respectively. The width of the peak is not given explicitly but can be calculated by subtracting the start time from the end time.

Following the peak description process, the Pulse Height Analysis (PHA) process takes the height of a positive peak, calculates the PHA channel that corresponds to that height and returns the PHA channel index.

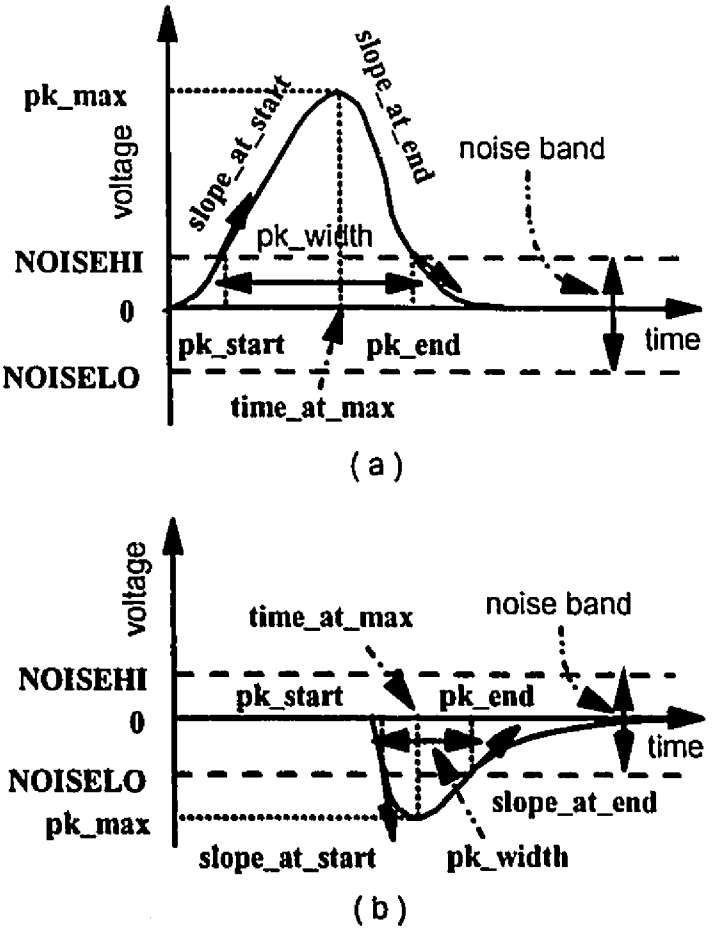

Figure 2.6 Peak Parameters: (a) positive peak, (b) negative peak

In total, six peak description parameters and a PHA channel number are processed. These seven parameters are encoded in order to decrease the number of data to be transferred and therefore increase the communication speed. Each peak is finally represented by a group of 16 bytes and a real-time data transfer process transfers the encoded data to the host PC. On the PC's side, a real-time DSP-PC interface receives and saves the data in memory.

2.3.3 DSP-host PC Interface

As mentioned before the DSP-host PC interface receives the encoded data from the DSP board and saves them into the PC's memory. In order to explain the operation of the interface, a brief description of the DSP56001 processor and DSP-56 board is needed.

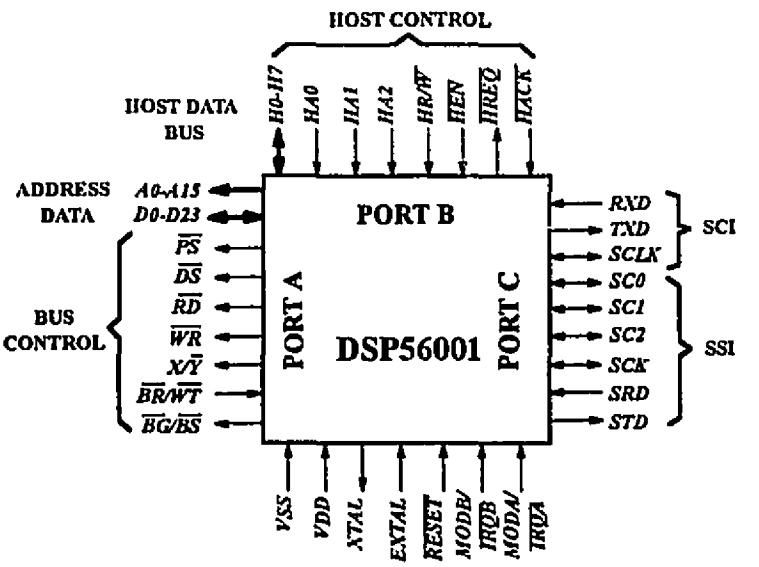

Figure 2.7 DSP56001 Functional Signal Groups

The functional signal groups of the DSP56001 chip are represented in Figure 2.7 and the DSP-56 board block diagram is shown in Figure 2.8 [Ariel 89].

Port B is a dual-purpose I/O port that can be used as 1) 15 general-purpose pins individually configurable as either inputs or outputs or as 2) an 8-bit bi-directional host interface (HIF) (Figure 2.7). In LiMCA operation, port B is configured as a host interface and provides a convenient connection to another processor.

The host interface is a byte-wide, full-duplex, parallel port that can be connected directly to the data bus of a host processor. The host processor may be any of a number of industry-standard microcomputers or MPUs, or another DSP. The DSP56001 host interface has an 8-bit bi-directional data bus, H0-H7, and seven dedicated control lines, HA0, HA1, HA2, HR/$\overline{W}$, $\overline{HEN}$, $\overline{HREQ}$, and $\overline{HACK}$, to control data transfers.

The host interface appears as a memory-mapped peripheral occupying eight bytes in the host-processor address space. Separate transmit and receive data registers are double buffered to allow the DSP56001 and host processor to efficiently transfer data at high speeds. Host processor communication via the host interface is accomplished using standard, data move instructions and addressing modes.

Port B is used as the real-time data passage between the host PC and DSP processes. Figure 2.8 is a block diagram of the DSP-56 hardware. The main subsystems

include: the DSP56001 processor, external data RAM, external program RAM, Analog I/O, SCSI Port, DSPnet Port, Auxiliary I/O.

The host interface is asynchronous and consists of two banks of registers - one accessible to the host processor and a second accessible to the DSP. The maximum data transfer rate at the host interface level is 8 Mbytes/s. There is also a difference on how the data are represented on the two sides of the host interface. On the DSP side the data

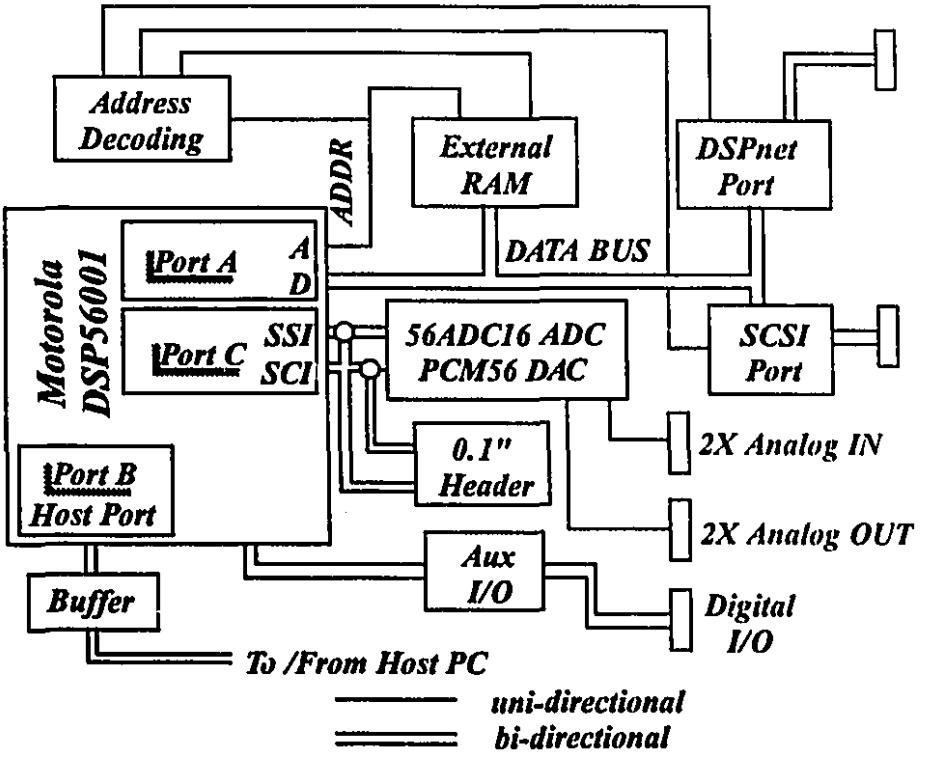

Figure 2.8 DSP-56 Block Diagram

word is 24 bits long and is mapped into three 8-bit I/O ports on the PC.

One peak is characterized by seven parameters as discussed in Section 2.3.2 and these parameters are encoded at the DSP level into 16 bytes. At the host PC level, a small program called *PkParmConvert* was written to acquire the encoded data and to decode the 16 bytes peak description into peak parameters (Appendix A).

One will notice that the data transfer rate and the amount of data transferred are factors affecting the communication between the DSP and the host PC. Another element that cannot be neglected is the maximum number of peaks that must be detected within a certain time period, i.e. 2000 peaks/s in the worst case. Considering the fact that each peak is represented by 16 bytes, 32 Kbytes of data are to be transferred per second. Therefore, efforts must be made to manage such a busy communication.

To be able to handle and to process all the data received from the DSP board, a higher level control program running at the host PC level has been developed. This program serves three purposes: one is to interface the PC with the DSP board, the second is to perform computational tasks, such as data calculation, manipulation, conversion, scaling, etc., and finally, the third purpose is to provide a user interface for the human operator. Because of the complex nature of the overall task, a friendly graphical user interface was conceived. In the next section, the design principles of graphical user interfaces will be presented.

2.4 Principles of Graphical User Interface Design

User interfaces are those parts of computing systems that allow the person using the computer to access the services offered. In other words, without user interfaces computers would be useless [Thimbleby 90].

The simplest view of human-computer interaction is shown in Figure 2.9. The arrow represents the user interface. The user interface is an information channel that conveys information between user and computer.

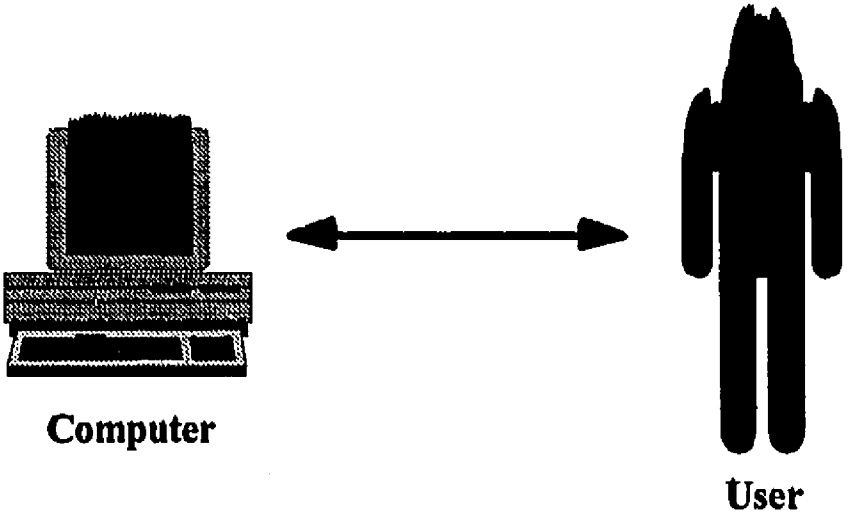

Figure 2.9 Computer-user interaction

From our perspective, however, Figure 2.9 is deficient because it omits the designer. Figure 2.10 rectifies this problem. First the designer implements a computer system (generally a software system, but sometimes the designer will be in a position to choose, if not influence, hardware aspects of the system): this is generally the result of intense work. Then the user interacts with the system, perhaps very intensely, over a much longer period. Although the interactions are represented with identical arrows, they represent very different styles of interaction.

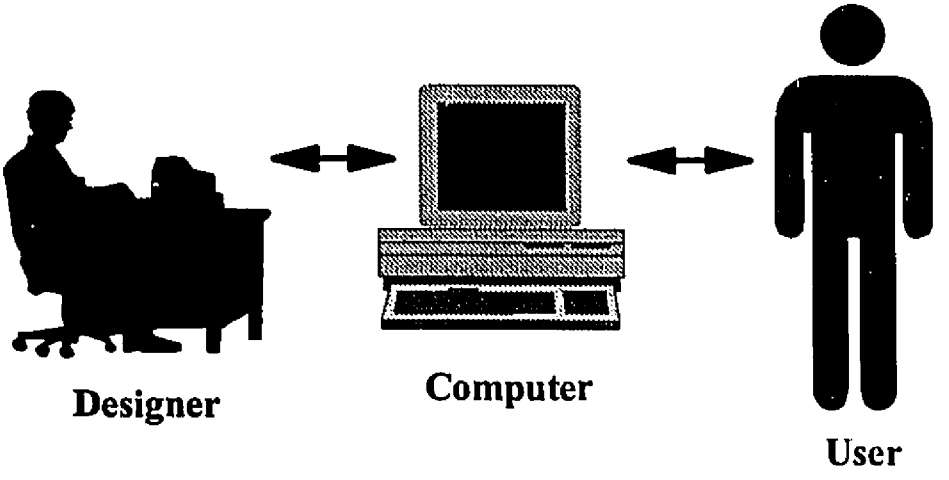

Figure 2.10 Bringing in the designer

The information flowing through the user interface is not sufficient to use the computer. There are information flows in addition to those shown that enable the user to operate. For instance, over the years, the user has acquired information about how the world and things in it operate and some of that knowledge must be drawn upon to use a computer. More specifically, the user will have information about the tasks he wants to undertake in conjunction with the computer system: these tasks will not be fully

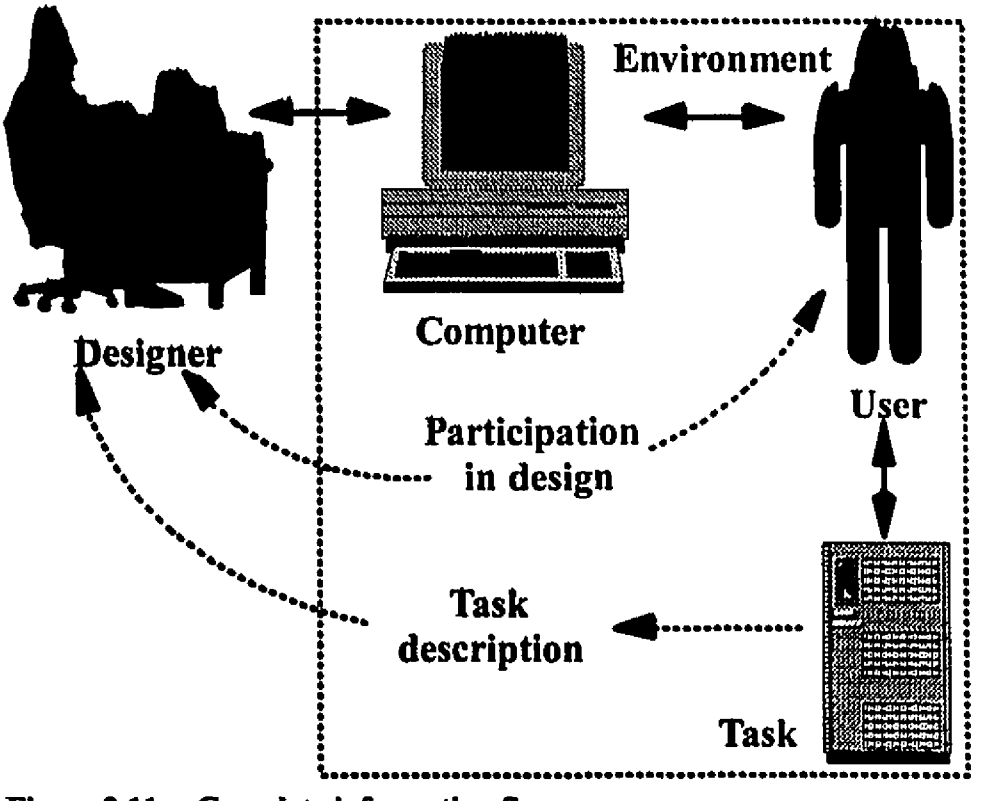

Figure 2.11 Complete information flow

represented in just the information flowing through the user interface, so a more realistic information-flow diagram is shown in Figure 2.11.

In typical industrial usage situations, cost shapes many judgments i.e., a lower cost solution may be preferred even if there is some sacrifice in reliability. Operator training time is expensive, so ease of learning is important. The tradeoffs for speed of performance and error rates are decided by the total cost over the system's lifetime [Shneiderman 92].

2.4.1 What makes a good design

Principles of design are hard to articulate: the more you state and use, the more exceptions there seem to be. Nevertheless, many user tasks or operations follow the same sorts of basic behaviors and can be modeled in similar fashions. The art and science of interface design depends largely on making the transactions with computers as transparent as possible in order to minimize the burden on the user.

Although there is no recipe for a good design, this section tries to present the underlying principles of design that are applicable to most interactive systems. These underlying principles of interface design, derived heuristically from experience, should be validated and refined:

- *Strive for consistency.* Consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus, and help screens; and consistent commands should be employed throughout.
- *Enable frequent users to use shortcuts.* As the frequency of use increases, so do the user's desire to reduce the number of interactions and to increase the pace of interaction.
- *Offer informative feedback.* For every operator action, there should be some system feedback.
- *Design dialogs to yield closure.* Sequences of actions should be organized into groups with a beginning, middle, and end.
- *Offer simple error handling.* As much as possible, the system should be designed so the user cannot make a serious error. If an error is made, the system should detect the error and offer simple, comprehensible mechanisms for handling the error.
- *Permit easy reversal of actions.* As much as possible, actions should be reversible. This feature relieves anxiety, since the user knows that errors can be undone.
- *Support internal center of control.* Experienced operators strongly desire the sense that they are in charge and that the system responds to their actions.

• *Reduce short-term memory load.* The limitation of human information processing in short-term memory requires that displays be kept simple.

These underlying principles must be interpreted, refined, and extended for each environment.

2.4.2 The design methodology

The first step in designing an interface is to decide what the interface must accomplish. Although at first this statement may seem overused, poor requirement definitions have marked numerous user-interface design projects at an early stage. Understanding user requirements can be accomplished in part by studying how the problem under consideration is currently solved. Another successful approach is for the designer to learn how to perform the tasks in question. The objective is to understand what prospective users currently do, and, more important, why they do it [Foley et al. 90].

When the requirements have been worked out, a top-down design is next completed by working through the four design levels: conceptual, functional, sequencing, and binding. The explanation for top-down design of user-interfaces is that it is best to work out global design issues before dealing with detailed, low-level issues.

The conceptual design is developed first and consists of the definition of the principal application concepts that must be mastered by the user, and is hence also called the *user's model* of the application. The conceptual design typically defines *objects*, *properties* of objects, *relationships* between objects, and *operations* on objects. In a simple text editor, for example, the objects are characters, lines, and files, a property of a file is its name, files are sequences of lines, lines are sequences of characters, operations on lines are Insert, Delete, Print, etc.

The functional design focuses on the commands and what they do. Functional design defines meanings, but not the sequence of actions or the devices with which the actions are conducted. Attention must be paid to the information each command requires, to the effects of each command, to the new or modified information presented to the user when the command is invoked, and to possible error conditions.

The sequencing and binding designs, which together define the form of the interface, are best developed together as a whole, rather than separately. The design involves first selecting an appropriate set of dialog styles, and then applying these styles to the specific functionality.

The whole design process is greatly helped by interleaving design with user-interface prototyping. A user-interface prototype is a quickly created version of some or all of the final interface, often with very limited functionality.

As mentioned in Section 2.3, the newly developed DSP-based LiMCA system has a complex nature and to ease the task of an operator the principles referred to above were used to design the user interface. Next chapter will be entirely dedicated to the presentation of the Graphical User Interface.

Chapter 3: GRAPHICAL USER INTERFACE IMPLEMENTATION

This chapter is dedicated entirely to the description of the Graphical User Interface designed for the DSP-based LiMCA developed at McGill University.

As mentioned in Section 2.3.3, a higher level control program running on the PC has been developed in order to interface down to the DSP board, to perform some computational tasks, such as data calculation, manipulation, conversion, scaling, etc., and to provide a user interface for the human operator. A friendly graphical user interface was developed to satisfy the complexity of the tasks listed above.

The operation of the LiMCA system involves four major steps: input of measurement-dependent parameters, system calibration, data acquisition and data analysis. Successful data acquisition and analysis depends on the correct selection of the parameters, which include operational parameters and physical properties of the metal. A large number of parameters need to be controlled or changed in order for the system to adapt to different operational conditions and different media. This parameter initialization introduces a big burden on the operator. To overcome all these inconveniences, a Graphical User Interface (GUI) is required. The objective of the GUI is to provide an "easy-to-navigate" environment. It offers very well organized windows with input fields, dialog boxes and graphical displays. For the multitude of parameters that are used, it is equally important that certain input range limitations and format controls to be applied to the input fields. In some cases, different fields need to be automatically disabled or enabled. Furthermore, unit selection and conversion are attached to each input field. Different templates and previous configurations can also be selected and retrieved. All of the above features facilitate the operation and help prevent input errors.

Due to the complex nature of a GUI, the design and implementation of such an interface is left to specialized programmers. Recent advances in software packages and programming methodologies made this job easier. For example, the Object Oriented Programming (OOP) technique made major contributions to the field of graphical user interface design.

As previously mentioned two commercial software packages, object-Menu from Island Systems and MetaWindow from Metagraphics Software Corporation, both based on the OOP technique, were used to implement the GUI. Borland C++ (version 3.0) software package was used as the basic compiler. During the development of the GUI,

we also adopted the OOP technique to implement the rest of the package, using the novel concepts specific to the OOP technique.

3.1 Software development environment

This section presents the environment in which the GUI was developed, and more specifically, the object-oriented programming technique and the software packages that were used as tools.

3.1.1 Object-oriented programming technique

There are five main kinds of programming styles, here listed together with the kinds of abstractions they use:

- Procedure-oriented — Algorithms
- Object-oriented — Classes and objects
- Logic-oriented — Goals, expressed in predicate calculus
- Rule-oriented — If-then rules
- Constraint-oriented — Invariant relationships

There is no single programming style that is best for all kinds of applications. For example, rule-oriented programming would be best for the design of a knowledge based system. The object-oriented style is best suited to the broadest set of applications, which includes industrial-strength software in which complexity is the dominant issue. By using object-oriented design, one creates software that is flexible to change and is written with economy of expression. In our design a greater level of confidence in the correctness of our software is achieved through an intelligent separation of its state space.

Figure 3.1 illustrates the topology of recently developed OOP languages as Smalltalk, Object Pascal, C++, CLOS, and Ada. The physical building blocks in these languages are the modules, which represent a logical collection of classes and objects instead of subprograms, as in earlier languages (e.g. FORTRAN). To state it another way, "If procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around the verbs while an object-oriented program is organized around nouns" [Booch 91]. For this reason, the physical structure of an object-oriented application appears as a graph, not as a tree, which is typical of algorithmically oriented languages. Additionally, there is little or no global data. Instead, data and operations are united in such a way that the fundamental logical buildings blocks of the system are no longer algorithms, but classes and objects.

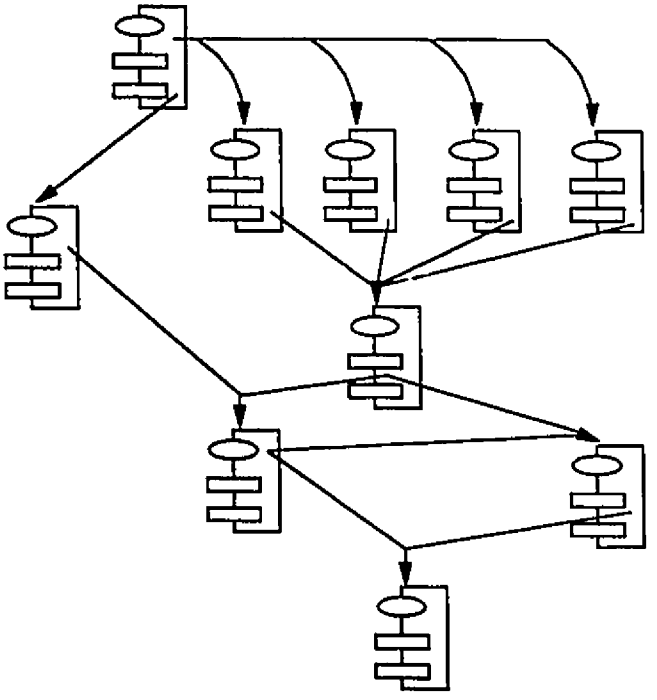

Figure 3.1 Topology of Applications using Object-Based and Object-Oriented Programming Languages

Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships. In such programs, classes are generally viewed as static, whereas objects typically have a much more dynamic nature.

There are three important parts in this definition: object-oriented programming (1) uses *objects*, not algorithms, as its logical building blocks; (2) each object is an *instance* of some *class*; and (3) classes are related to one another via *inheritance* relationships.

In the object-oriented context, the conceptual framework is the *object model*. There are four major elements of this model:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries,

relative to the perspective of the viewer. An abstraction focuses on the outside view of an object, and thus serves to separate an object's essential behavior from its implementation.

Abstraction and encapsulation are complementary concepts: abstraction focuses upon the outside view of an object and encapsulation prevents clients from seeing its inside view, where the behavior of the abstraction is implemented. In this manner, encapsulation provides explicit barriers among different abstractions. In practice, each class must have two parts: an interface and an implementation. The interface of a class captures only its outside view, encompassing our abstraction of the behavior common to all instances of the class. The implementation of a class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behavior. To summarize, encapsulation is the process of hiding all the details of an object that do not contribute to its essential characteristics.

Modularity is the property of a system that has been decomposed into a set of coupled modules. Modules in C++ are nothing more than separately compiled files. The traditional practice in the C/C++ community is to place module interfaces in files named with a *.h* suffix; these are called *header files*. Module implementations are placed in source code files distinguished by a *.cpp* suffix. Dependencies among files can then be asserted using the #include preprocessor directive. This approach is one of convention; it is neither required nor enforced by the language itself.

The concept of modularity is important in the LiMCA implementation because of two reasons: (1) since modules serve as the elementary and indivisible units of software that can be reused across applications, a developer might choose to package classes and objects into modules in a way that makes their reuse convenient;

(2) many compilers generate object code in segments, one for each module. This places practical limits on the size of individual modules.

A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our design, the understanding of the problem is greatly simplified. Thus a hierarchy can be defined as a ranking or ordering of abstractions.

Inheritance is the most important "kind of" hierarchy and it is an essential element of object-oriented systems. Basically, inheritance defines a relationship among classes, wherein one class shares the structure or behavior defined in others. Inheritance thus represents a hierarchy of abstractions, in which a subclass inherits from one or more superclasses. Typically, a subclass enlarges or redefines the existing structure and behavior of its superclasses.

Different programming languages trade off support for encapsulation and inheritance in different ways, but C++ offers the greatest flexibility. Specifically, the interface of a class may have three parts: *private*, which declare members that are visible only to the class itself, *protected*, which declare members that are visible only to the class and its subclasses, and *public*, which are visible to all.

I mentioned several times the terms object and class. Therefore, a short description of this terms is necessary.

One can informally define an object as a tangible entity that exhibits some well-defined behavior. From our perspective, an object is any of the following: a tangible and/or visible thing, something that may be understood intellectually, and something towards which thought or action is directed. Some objects may have crisp conceptual boundaries, yet represent intangible events or processes. Other objects may be tangible, yet have fuzzy physical boundaries. One can conclude that an object has state, behaviour, and identity, and that the structure and behavior of similar objects are defined in their common class.

The concepts of a class and an object are tightly interconnected, for one cannot talk about an object without reference to its class. However, there are important differences between the two terms. Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction. A class is a set of abstract objects that share a common structure and a common behavior. A single object is simply an instance of a class. Whereas an individual object is a concrete entity that performs some role in the overall system, the class captures the structure and behavior common to all related objects.

As mentioned earlier, the interface of a class can be divided into public, protected, and private parts. The C++ language does the best job among object-oriented languages in allowing a developer to make explicit distinctions among these different parts of a class's interface.

Programming using objects is fundamentally different than the models embraced by the more traditional methods of structured programming. The use of the object model leads us to construct well-structured complex systems.

3.1.2 MetaWINDOW

MetaWINDOW is a professional graphics programming toolkit that integrates a comprehensive set of drawing routines and dynamic runtime support for a broad array of graphics devices into a highly unified graphics development system. MetaWINDOW

includes a powerful set of drawing functions and an expanded set of advanced utilities for developing multi-window desktop applications.

MetaWINDOW's graphics facilities are suited for all PC-based graphics applications. Between the features not found in other graphics systems, one can mention:

• *Open interface*: gives the user direct access to all levels of the graphics system. This open architecture improves performance using less memory, thus making the code more efficient.

• *Virtual bitmaps*: in addition to the default screen bitmap, the user can also define any number of off-screen "virtual bitmaps". Virtual bitmaps can be of any size and located in local memory, EMS memory, XMS memory or disk-cached virtual memory buffers.

• *Optimizing graphics executive*: the executive system links only those functions and drawing attributes used by the program. In addition, the executive dynamically loads and links graphics drivers at runtime, keeping code size to a minimum.

• *Efficient event-driven interface*: MetaWINDOW includes an enhanced event driven user-interface that ensures that time critical user services, such as mouse/cursor tracking, keyboard, function key and mouse button input selections, are processed as asynchronous priorities automatically by the MetaWINDOW system. These built-in services eliminate the need for time dependent "polling" loops within the application program, and insure that operator inputs are serviced even if the application is performing extended calculations or I/O processing.

• *Multiple ports and windows*: the open interface design is very well suited for multi-window applications. Using the "port" structures, one can create multiple graphics windows, where each window is treated as a separate and independent display screen.

• *Object select tests*: MetaWINDOW includes an advanced set of object select functions that allow the user to find if a user input selection is "in", "on" or "outside" a particular graphics object.

There are several basic items needed for a MetaWINDOW-based program. This consists of:

1) An "include" statement for MetaWINDOW header files.

A program must contain an "include" statement referencing a master MetaWINDOW header file, **METAWNDO**. The master header file in turn references several other key MetaWINDOW header files: **METCONST** - Graphic structures and constants, **METPORTS** - Bitmap and port data structure definitions, **METEXTRN** - Function call prototypes, and **METFONTS** - Font file data structure.

The MetaWINDOW header files define the basic data structures, constant and function prototypes the user needs to use.

2) MetaWINDOW initialization.

Before starting the drawing calls, the user must first initialize the MetaWINDOW system, and then switch the display hardware to graphics mode. The **InitGraphics()** procedure initializes the MetaWINDOW system, and graphics procedures for a particular video adapter and display mode.

3) Switching to graphics mode.

The **InitGraphics()** function initializes only the MetaWINDOW software system. It does not touch the display hardware or switch the screen into graphics mode. Procedure **SetDisplay()** is called to physically switch the graphics adaptor hardware between graphics and text modes.

4) Drawing objects to the screen.

Once in graphics mode, objects such as lines, rectangles, ovals, text and other graphics can be drawn to the screen. An X,Y Cartesian coordinate system is used to reference positions on the display screen. The "global" (0,0) origin of the coordinate system is typically located in the upper-left corner of the screen, with positive X values increasing to the right, and positive Y values increasing downward towards the bottom of the screen.

5) Switching back to text mode.

As mentioned above, procedure **SetDisplay()** is called to switch between modes.

6) Terminating MetaWINDOW.

One must tell MetaWINDOW when to terminate the program (the Graphics subsystem is left at a known state when the program terminates). This is performed using the **StopGraphics()** function call. Calling **StopGraphics()** at the end of the program is important, because MetaWINDOW links internally the system at a very low level and needs to detach itself prior to exiting the program.

One of the important features of the MetaWINDOW software package is the event system. The event system ensures that operator input and tracking actions are serviced immediately and represents the backbone of a modern graphical user interface. It allows cursor tracking, keyboard input, and mouse button selections to proceed even when the application program is processing actions. These operator "events" are stored into a circular "event-queue" buffer for servicing sequentially by the application program.

The event queue operates similar to a keyboard type-ahead buffer. When a program operation completes, control is passed back to the main event loop which checks to see if there is another event to process. If an event is posted, the program dispatches to the appropriate function to handle the event.

With event driven applications, a main event loop retrieves messages stored in the event queue, and executes the appropriate function to process the input action. If no events are pending, the application can optionally perform background processing while it waits for the next command. Time consuming functions, such as saving data files or processing screen updates, can be relegated as background operations. In addition to operator input events for mouse and keyboard actions, function **StoreEvent()** can be used to send "program events" through the event queue.

3.1.3 object-Menu

The object-Menu package is a comprehensive, object-oriented GUI toolkit. Its library elements provide a state-of-the-art graphical user interface that can be integrated into any C++ application. object-Menu takes advantage of the capabilities of C++ to provide extensive functionality in a compact format. Some of the features of this software package are:

- *Event driven*: complete event system handles menu and window events as well as user definable events to facilitate creation of a multi-tasking system.
- *Mail system*: messages can be sent from one event (window, menu, user defined event, etc.) to another to enable an interaction between system elements.
- *Object-oriented implementation and usage*: object-Menu is fully object-oriented and takes full advantage of OOP. The use of extensible classes means that menu types can be inherited and enhanced to provide a high degree of control to a C++ user, without modifying the sources.

Components of the object-Menu package include windows, menus, dialogue boxes, data entry and text edit and display blocks. Special features of object-Menu components are as follows:

- Menus

Menu orientation can be horizontal or vertical. Menus can be attached and aligned or "popped up". Menus can include buttons, check marks, and user defined icons.

- Windows

A complete window system is maintained to deal with window events and overlapping windows. The underlying windows can be restored quickly with minimal memory utilization and without regenerating the image.

- Scroll bars

Various scroll modes can be set to allow scrolling at finer levels of resolution and the scroll architecture facilitates several advanced scrolling mechanisms.

- Data Entry

The edit capability combined with picture strings create powerful data entry functionality that can be used for application input. The data entry field can be mixed with different font styles, sizes or colors to create a custom look. Data entry elements can be easily mixed with other elements in a dialog box.

• Dialog Boxes

Combinations of all menu and data entry elements can be mixed in a dialog box to logically group input selections.

As mentioned earlier in this chapter, object-Menu includes an event manager for dealing with all input events and real-time tasks. There are two distinct event systems. The primary event system deals with all input event handling. The user-task event system provides means for running real-time background tasks such as for example, a task to poll a communication channel.

The event system passes user input to the appropriate event handler (menu, window, or other) for processing. Each event runs independently of the rest of the system. An **omEventQueue** represents one cycle or group of events. The entire system is made up of several omEventQueue's organized in a tree-like structure. The method functions of an omEventQueue coordinate the passing of control between omEventQueue's and to each individual event handler.

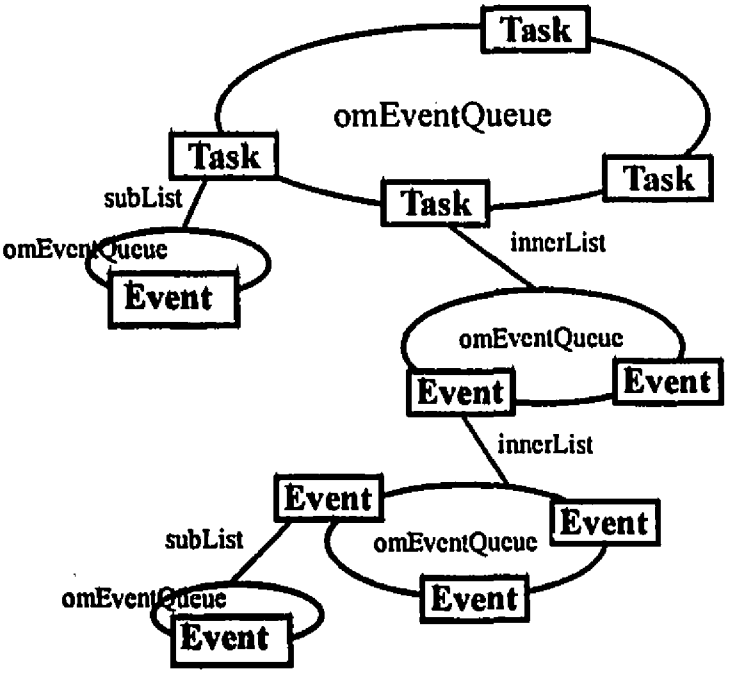

Figure 3.2 Conceptual view of the event tree

As described above and illustrated in Figure 3.2, several omEventQueue's are combined in a tree structure to create a complete representation of all events in the system. At the highest level there is the main event queue organized as a loop of several events. Each event can point to two additional event queues: a **subList** and an **innerList**. Normally only one of the two pointers will be used for a particular event. The subList points to a queue that runs a submenu. This queue will always contain only one event.

The innerList points to a queue of attached elements. For example, the inner elements of a dialog box or a window are in its innerList. Since each component of the innerList is an event and can point to an attached innerList, a tree structure of all the event queues is formed.

The concept of an innerList is central to the event tree structure in the object-Menu system. Each of the items in a dialog box or window type element is placed on the omEvent Queue pointed to by the innerList of the item. Both the parent item and the innerList items have their own event handlers. When the parent event handler determines that the current input is intended for one of its innerList elements, it indicates to the event manager that control should be passed to its innerList. This causes the omEventQueue of the innerList to run independently of the rest of the system.

The heart of the event system is the individual event handler. The event handler is a virtual function called **heyUser**. The job of the heyUser is to perform the appropriate action based on the input and then return a status code to the queue manager to indicate if this event should remain active. The heyUser is only called if the event is active and enabled. Note that when an input activates an event, that input will always be passed first to the heyUser before a new input is solicited from the input queue.

After presenting the environment in which our software was created, the next part will describe the architecture and the features of the DSP-LiMCA software.

3.2 Software architecture

The GUI is structured in three "layers": the top one is a group of windows that provide different input fields, a real-time graphic display and an analysis display. The second layer includes all the background computational tasks (such as parameters setup, file management, unit conversion, axis scaling, peak classification, peak parameter decoding, PHA, MCS, calibration, etc). The third layer is the DSP-host PC interface and contains such routines as the DSP driver down-loading process, the DSP command down-loading process, the real-time data transfer, the DSP memory read and write, and the expanded memory manager. As one can notice in Figure 3.3, the middle layer performs

most of the calculations for the LiMCA processes and has bilateral communication with the front and bottom layer. The algorithms that realize LiMCA processing are implemented in this layer by separate modules. Object oriented programming techniques were used to make these modules encapsulated, reusable and efficient.

For the bottom layer, the main concern is the speed of communication. Therefore, most of the modules in this layer are interrupt service routines which ensure the time of response. Also, the number of the levels of function calls and switch commands is limited. In-line assembly language is used because it can be precisely timed and proper timing is needed in a real-time application. In this layer the computations are minimized and most

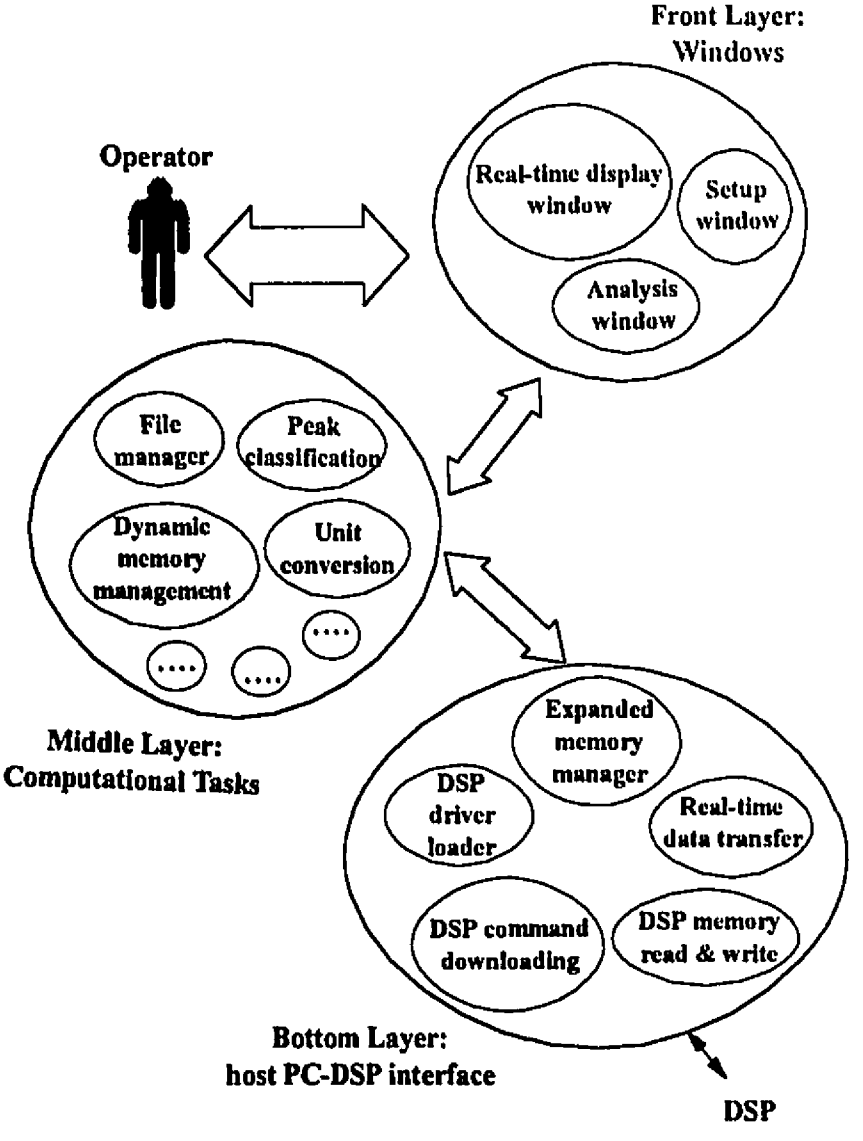

Figure 3.3 The GUI structure

of the data processing tasks is left for the middle layer. The main data link between this layer and the middle layer is the expanded memory. Therefore, an expanded memory managing module is also implemented in this layer. All the above measures help the host PC to catch up with the high speed DSP process preventing an eventual DSP overflow. The implementation of the first layer software will be discussed later.

To implement the GUI, the data involved in the whole process must first be analyzed and structured. The background tasks must also be coded in parallel with the front windows.

The input parameters have been organized into several groups:

1. *General parameters*: test name, test directory, test location, test number, acquisition number, test date, test time, and data file names (setup data file, acquisition data file, MCA data file, calibration data file). These are the parameters describing the general information about a test.
2. *Physical properties*: medium type (water or metal), medium, density, resistivity, and discharge coefficient of medium. All the physical properties about the medium to be tested are included in this group.
3. *Test conditions* are listed in this group as: sensor type, orifice diameter, ESZ current, differential pressure, immersion depth, and working temperature.
4. *DSP parameters*: channel model (mono or stereo), channel, sampling frequency, high noise level, and low noise level. These parameters are used to configure the DSP process.
5. *Other parameters*: test mode (PHA or MCS), acquisition time, MCS time increment, MCS number of scan, minimum size to be detected, number of MCA channels, number of size ranges to display, and size ranges to be displayed.

All the input parameters are saved in a Setup class in memory when the program is running and also saved on the hard-disk in a group of files. These files are managed by the user interface and are invisible to the operator. In this way the operator can access and modify the data only through the user interface. The rules and restrictions imposed on each field by the interface guarantee the correct data format and input range. Also these files can always be retrieved for purposes of analysis.

The peak description parameters are uploaded from the DSP board to the host PC through an interrupt driven mechanism. At the host PC level, these parameters are first directly saved in expanded memory in real-time. Between the real-time data transfer cycles, the peak parameters are decoded, processed, displayed, and saved in the acquisition data file inside the hidden file structure.

The source code of the LiMCA software is structured in multiple modules. A list of these modules can be found in Appendix B. Each module contains a group of functions and classes. The global variables, class definitions, and constants are defined in the header (*.h*) files and the module implementations are located in files named with a *.cpp* suffix, required by the Borland C++ compiler.

The modular structure is dictated by the size of the source code, that is the variables cannot be compiled because they exceed the capacity of the compiler. In order to compile several source files, each of which may need to pass through preprocessors, assemblers, and compilers, a Makefile was used. For details on the compilation and linkage processes, see Section 3.5.

3.3 The front layer software (GUI)

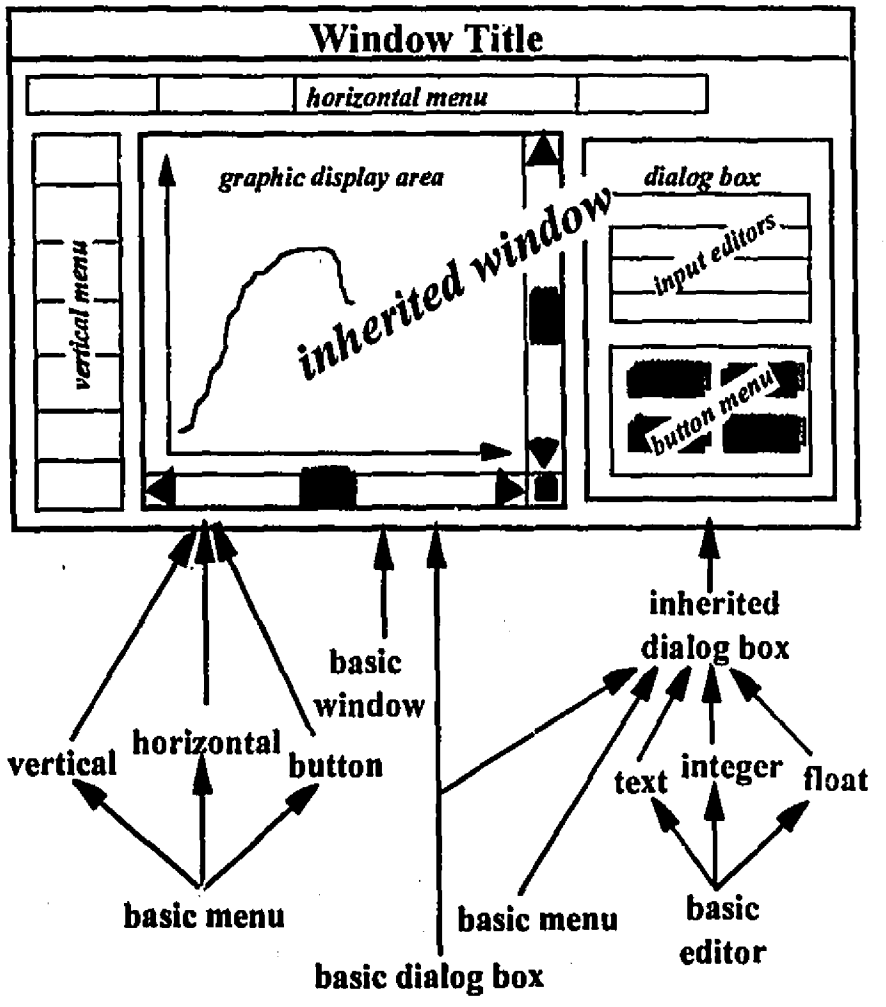

Figure 3.4 The inheritance concept design of a window

The front layer contains three major windows: the setup window, the real-time display window and the analysis window. Each includes a selection of sub-menus (horizontal, vertical, and button menus), dialog boxes, graphic display windows, and data display windows. The basic design of a window for the GUI is illustrated in Figure 3.4.

The basic building blocks of a complex window (i.e. the "inherited window" in Figure 3.4) are the basic classes such as the basic menu classes, the basic editor classes, the basic window classes, the basic dialog box classes, and other inherited classes. All the basic classes were available with the commercial software packages. The inherited classes

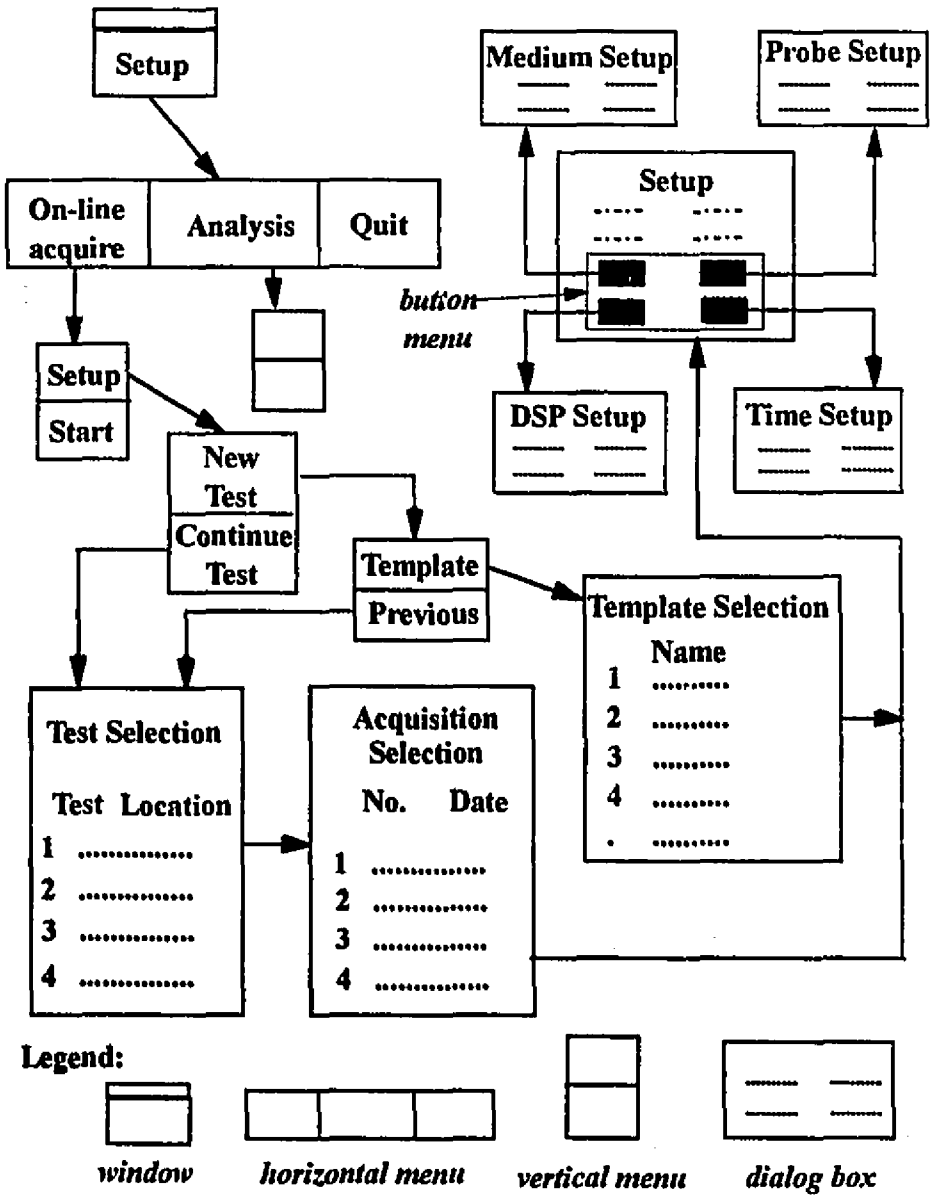

Figure 3.5 The hierarchical structure of the setup window

were built subsequently. Once an inherited class had been defined, it was treated as a basic class which could be reused without repeated coding. Using the class inheritance technique, our own window objects were inherited from a number of other inherited classes and basic classes, and were thus built up efficiently.

Figure 3.5 demonstrates the hierarchical structure of the setup window. This window is inherited from a basic window class, several levels of horizontal menu classes, vertical menu classes, and several dialog box classes. These menu and dialog classes are themselves inherited from basic menu classes and basic dialog box classes. In particular, one can notice that the setup dialog box and its four "child" dialog boxes became the basic classes in the construction of two other dialog boxes. They are the "Template selection" and the "Acquisition selection" dialog boxes. One can also notice that the "Test selection" dialog box, which is the parent dialog box of the "Acquisition" dialog box, is itself the child object of two vertical menus. In this way, the addition of new objects, the modification and the maintenance of present objects, are easy to carry out. The effort to add new applications based on previously written classes is minimized because of the reusability and portability of these classes. Therefore, this programming methodology suits on-going projects, in which frequent improvements cannot be avoided.

3.4 Software Design & Implementation

The new DSP-based LiMCA is completely different conceptually than the previous LiMCA generation and for this reason an entirely rebuilt interface was needed. This goal was achieved by analyzing the description of the user's actions from the user's point of view and combining it with the characteristics of use based on real work situations.

At the start of the design process, a general structure for the interface was decided, based on the previous experience with the LiMCA system. This structure was designed during several group meetings and was improved by taking into account the feedback from the users of the first versions of the newly designed interface. Two important user requirements needed to be met: first the system has to be configurable, i.e. all the parameters of the LiMCA system must be initialized appropriately, and second a real-time display is needed to show graphically the different distributions important to a metallurgist.

Due to the fact that this DSP-based LiMCA system was built for research purposes, different material configurations have to be available for the researcher. This is one of the advantages of our system compared with the commercially available LiMCA version built by BOMEM Inc. Another benefit consists in the fact that the system is

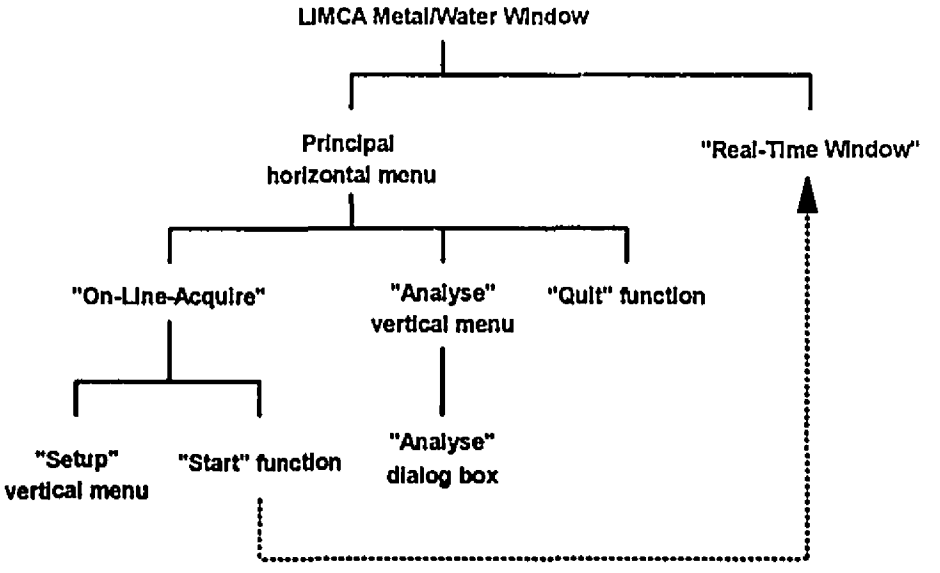

Figure 3.6 User Interface Structure - 1

"open" to future developments, that is the object-oriented software is flexible and enables the addition of new features without changing the previously written code.

The structure of the user interface is presented in Figures 3.6 through 3.10. All these figures are related to each other in the sense that the description starts with Figure 3.6 and continues in a tree-like structure. Figure 3.6 illustrates the root of the tree, the other ones being the branches of the tree.

Figure 3.6 represents the starting point in the user interface. The first objective in the design was a simple start screen. The LiMCA Metal/Water Window provides the Principal Horizontal Menu through which the user can perform the setup of the system in order to start the operation. Through the Principal horizontal menu the user has different choices: start acquiring data, analyze data already acquired or quit the program.

The "On-Line-Acquire" gives the possibility to input the parameters needed for the experiment or to start the acquisition and, in this way, to see the actual distribution in the real-time window. In order to start the acquisition, the user can choose in the "Setup" vertical menu between starting a "New test" and continuing a test (see Figure 3.7). The item "Continue test" was incorporated because there is a strong possibility that the operator needs to stop a test. In this way, the respective test can be continued from the point it was stopped. If "Continue test" is selected, the "Test Selection" dialog box comes

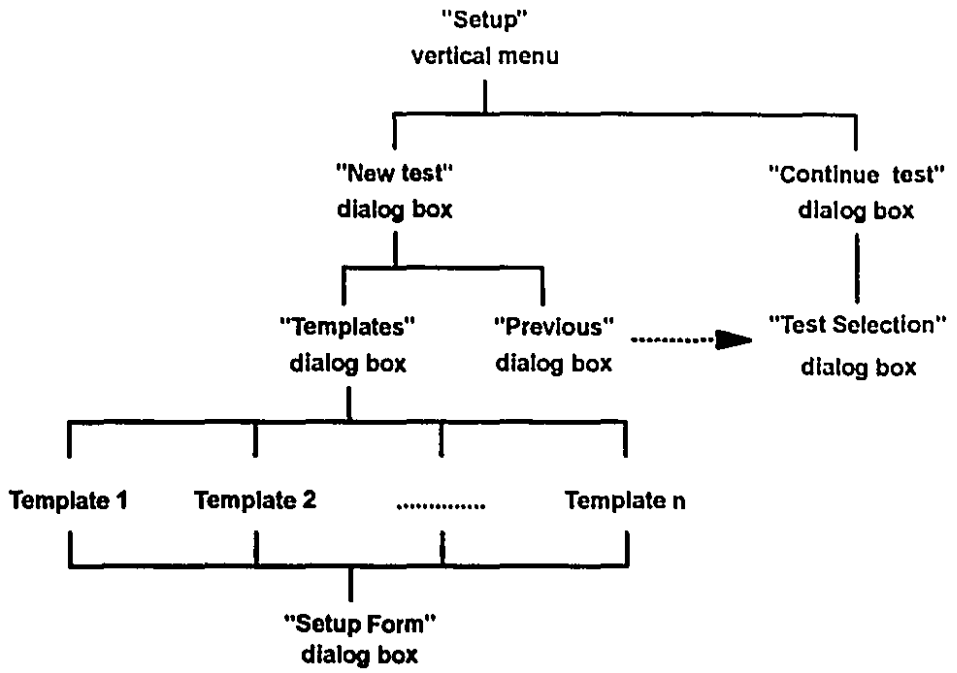

Figure 3.7 User Interface Structure - 2

up on the screen and the operator can select one of the previously started tests that have been saved into memory.

If a "New" test is picked from the "Setup" vertical menu, the operator can select between using one of the several available templates or using a previous test. The templates contain default values for different types of media and can be used as a point of departure in the respective test. The "Previous" acquisition feature was included to enable the operator to reuse some of the elements of previous tests and, in this way, to save time and to eliminate the possibility of data entry errors.

No matter what selection is made from the "Setup" vertical menu, the user ends up in the "Setup Form" dialog box which is illustrated in Figure 3.8. This dialog box is the central point of the software and provides the user with the means of selecting and inputting fields for all the parameters of the LiMCA system. Because of its importance, and to enforce the statements made in Section 3.1.1, the implementation of the "Setup Form" dialog box will be presented next as a representative example for the whole software package.

The length of the software makes impossible to present it entirely, but by examining this illustrative part and by considering that all the other defined classes are similar, one should understand the application quite clearly.

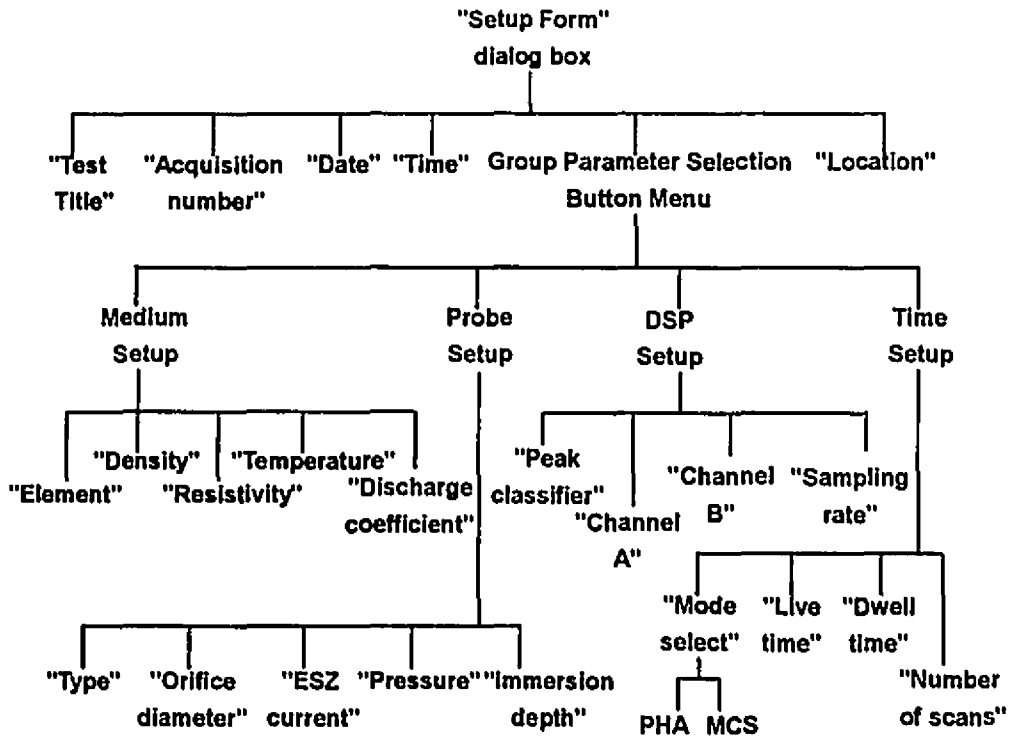

Figure 3.8 User Interface Structure - 3

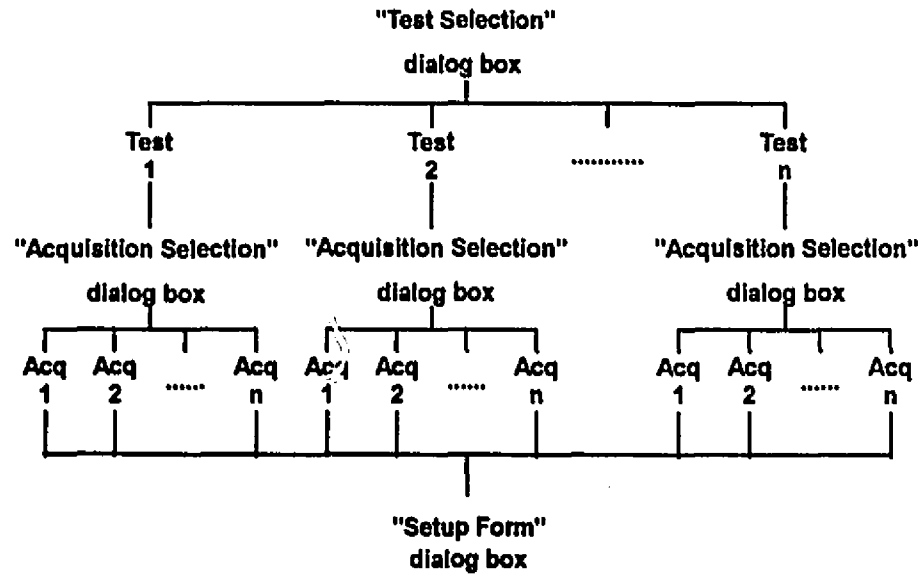

Figure 3.9 User Interface Structure - 4

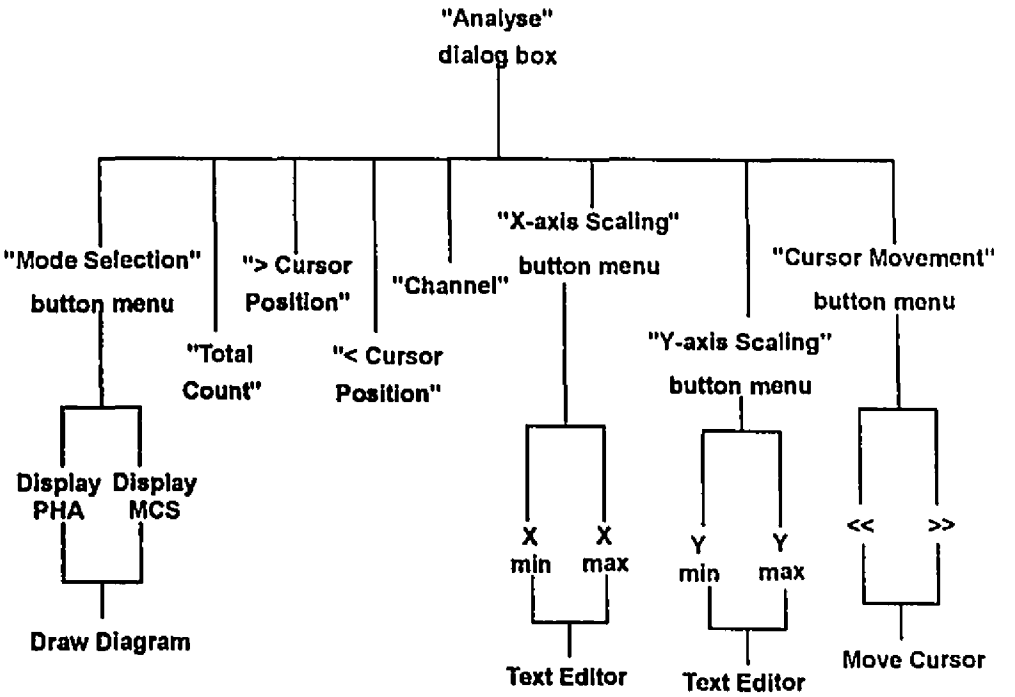

Figure 3.10 User Interface Structure - 5

Before describing the structure of the "Setup Form" dialog box, it is worthwhile mentioning that Figure 3.9 logically follows Figure 3.7 and Figure 3.10 succeeds Figure 3.6. Figure 3.9 summarizes the "Test Selection" procedure and Figure 3.10 describes the "Analyse" dialog box which enables the user to observe the distribution that constitutes the final result of the experiment. As one can notice in Figure 3.10, the user has the possibility to change the scaling and the mode selection, to move a cursor across the diagram and to monitor different useful values on the text editors.

To implement a class, one has first to define it and then the interface functions have to be implemented. The way the program is organized is the following: the definitions are given in the files with a .H suffix and the implementations in the files with a .CPP suffix.

The definition of the class called `SetupFormDialog` is given in the header module `M3.H`. The header file contains the declaration of the class `SetupFormDialog` as a derived class of the base class `omDialog`. `omDialog` is a dialog box class contained in the object-Menu commercial package. A dialog box is a collection of menu objects that are automatically aligned and enabled to interact as a unit. This means that the event handler treats the dialog box as a single selectable entity until the user enters it.

At that point, it is said to be "focused" and each of the dialog's components can be individually selected and run.

```
//part of the header file m3.h
class SetupFormDialog : public omDialog {
    public:
     myMetalDialog *md;
     myProbeDialog *pd;
     myDSPDialog *dd;
     myTimeDialog *td;
     omLabel *l1, *l2, *l2a, *l3, *l3a, *l4, *l4a, *l5;
     myTextEditor *te1, *te5;
     omButtonMenu *bm, *ok;
    //constructor
    SetupFormDialog( void );
    omBoolean doModalExit( void );
    };
```

The class `SetupFormDialog` is declared as a public member of the base class and inherits all the members of the base class `omDialog`. Public specifies that the public components of class `omDialog` remain public when regarded as components of class `SetupFormDialog`. The derived class `SetupFormDialog` is similar to its base class `omDialog` , but has additional function components beyond those of the base class and some `SetupFormDialog` member functions differ from those of the base class.

After the definition of the class, a number of variables are defined as public members of the class. These public members can be viewed as two groups. One is a group of pointers to several other dialog boxes. These pointers link the "child" dialog boxes like described in Figure 3.5. These dialog boxes are `myMetalDialog`, `myProbeDialog`, `myDSPDialog`, and `myTimeDialog` and provide input fields classified as in Section 3.2. The other group also contains pointers, but these ones are for input and display fields inside the principal dialog box (see Figure 3.5). These fields correspond to the labels used for titles and subtitles, to the text editors used to input certain data, and to the button menus used to access the other dialog boxes and to confirm that the setup window is correctly completed. After the definition of the pointers, the constructor of the class and an interface function are defined.

The constructor and interface functions for the class `SetupFormDialog` are included in the file `M1.CPP` which is attached in Appendix C. The constructor has to carry out five tasks to complete the construction of the class. The first task refers to the

input fields, then memory has to be allocated for the four "child" dialog boxes. The next task consists in defining a button menu, linking each button to the "child" dialog boxes, and putting them in the local event queue. A suitable exit function has to be implemented and this represents the fourth task and, finally, all these processes have to be included in a last event queue.

The structure of the input fields is represented in Figure 3.11. The format was chosen in this way because of the nature of the parameters. Some fields require text input, some decimal numbers or integer numbers. The value of some parameters is dependent on

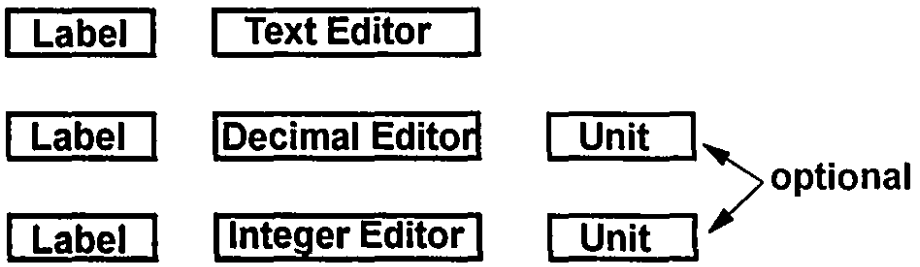

Figure 3.11 Format of an input field

the chosen unit. This is the reason why three types of editors had to be implemented as basic construction elements for input fields. Moreover, a pull-down vertical menu with scroll bar was implemented for the units option to provide the user with the possibility of selecting the units with which he is familiar.

Three classes, `myTextEditor`, `myDecimalEditor`, and `myIntEditor`, were implemented in the module `M1.CPP` to respond to the above mentioned purposes. All these classes are built based on the `omLineEditor` class provided in the objectMenu software package. This line editor class can be used to allow the user to enter any textual information and can also be used as a data entry editor for use in a simple or complex multiple field form. The definitions and the body of the three classes are given in Appendix D.

In the source code, the fields are numbered from 1 to n. Letter `l` is used to represent a label, `te` to represent a text editor, `de` to represent a decimal editor, and `ie` to represent an integer editor. Hence for example `te3` means the pointer pointing to the object that represents a text editor of the third field of the dialog box.

The second task consisted in allocating memory for the four "child" dialog boxes. These dialog boxes are defined in the header file `M3.H` and this part is included in Appendix E. As one can notice in Appendix C, memory is now allocated for each of the four dialog boxes: medium, probe, DSP, and time. The pointers used for each dialog box

are designated respectively by the names md, pd, dd, and td. The `new` operator offers dynamic storage allocation, similar but superior to the standard library function `malloc`. `new` tries to create an object of type `myMetalDialog` (or `myProbeDialog`, or `myDSPDialog`, or `myTimeDialog`) by allocating `sizeof(myMetalDialog)` bytes in the heap. The storage duration of the new object is from the point of creation until the operator `delete` kills it by deallocating its memory, or until the end of the program. If successful, `new` returns a pointer to the new object. A null pointer indicates a failure (such as insufficient or fragmented heap memory).

The third task included the definition of a button menu called bm and allocating memory in the same way as above. A button menu is an array of menu items. The default array configuration is one row for as many columns as there are items, but any row/column combination can be specified. A four-button menu has been chosen for our application in a two rows two columns configuration. The four buttons are called respectively "Medium_setup", "Probe_setup", "DSP_setup", and "Time_setup" and can be selected either by clicking the mouse on them or by using a hotkey, i.e. by pressing the key ALT and the one of the letters M, P, D, or T.

After implementing the button menu which enables the user to set the parameters through the "child" dialog boxes, another two buttons were programmed to offer a convenient method for accepting or discarding the information introduced by the operator. These two buttons are labeled "OK" and "CANCEL". By pressing or clicking with the mouse on the "OK" button, the respective form is accepted. The "CANCEL" button brings the user back to the beginning of the form and the enables him to start all the process again.

Finally, the last task was to link all the labels, editors, and button menus into an event queue. In order to place this event queue as an event in a parent queue, a pointer to this whole queue should be returned to the parent queue. Thus the parent queue can access and manage the local event queue. In our case the "Setup Form" dialog box is used by two other dialog boxes as in Figure 3.5. These two dialog boxes are "Template Selection" and "Acquisition Selection". These dialog boxes are treated by "Setup Form" dialog box as parent event queues and the pointer to it is returned to the parent queue by "this" when the construction of the object is completed.

In this way different levels of event queues are constructed and the link between levels are pointers. Each of the pointers is initiated by its constructor and returned through the "this" pointer.

The whole program is a group of queues like the one presented above linked together in a structure but at different levels. The top level is a queue called the "root

queue" which in our case is a principal window with horizontal and vertical menus as presented in Figures 3.4 and 3.6. The source code of the main program is included in Appendix F.

This software, when started, constructs all of the event queues until the top, and each event queue is constructed in the same way as the example presented here (the same process applied for all the queues). The construction process starts from the bottom level of the Medium, Probe, DSP, and Time dialog boxes, all the way up to the principal window and this window is pointed by the omMEQ pointer which is specifically defined by objectMenu as a pointer to the root queue. After the construction of the event queues, the program enters in a stage of execution in which each event can be activated in response of the operator's choices. The activated or focused event can "travel" around the queue structure constructed in the first stage.

As mentioned in Sections 3.1.1 and 3.2, the source code is organized in multiple modules. All the global variables, prototypes, and constants are defined in the header files and the source code was split between nine `*.CPP` files. To compile and link all the header and source files along with objectMenu, MetaWINDOW, and Borland C++ libraries is a process that requires many steps. Therefore, it is worthwhile to present the procedure of compilation and linkage. The main utility used is the MAKE utility provided in the Borland C++ compiler.

3.5 Compilation & linkage

MAKE is provided with a description of how the source and object files of the program are processed to produce the finished product. MAKE looks at that description and at the date stamps on the files, then does what's necessary to create an up-to-date version. During this process, MAKE invokes the compiler, the linker, and the utilities, but it never does more than is necessary to update the finished program.

MAKE keeps the program up-to-date by performing the following tasks:

- Reads a special file (called a makefile with the name lmc.mak). This file tells MAKE which .OBJ and library files have to be linked in order to create the executable file, and which source and header files have to be compiled to create each .OBJ file.
- Checks the time and date of each .OBJ file against the time and date of the source and header files it depends on. If any of these is later than the .OBJ file, MAKE knows that the file has been modified and that the source file must be recompiled.
- Calls the compiler to recompile the source file.

• Once all the .OBJ file dependencies have been checked, checks the date and time of each of the .OBJ files against the date and time of the executable file.
• If any of the .OBJ files is later than the .EXE, calls the linker to recreate the .EXE file.

The process of obtaining the executable program is represented in Figure 3.12. The source code is first written as a text file or as several modules. The source code is compiled with the Borland C++ compiler and the corresponding object files are obtained. After this stage, the linkage process follows and consists in combining the object files with the different libraries used, in the present case the Borland C++, the METAWINDOW, and the object MENU libraries. In this way, the executable program is obtained.

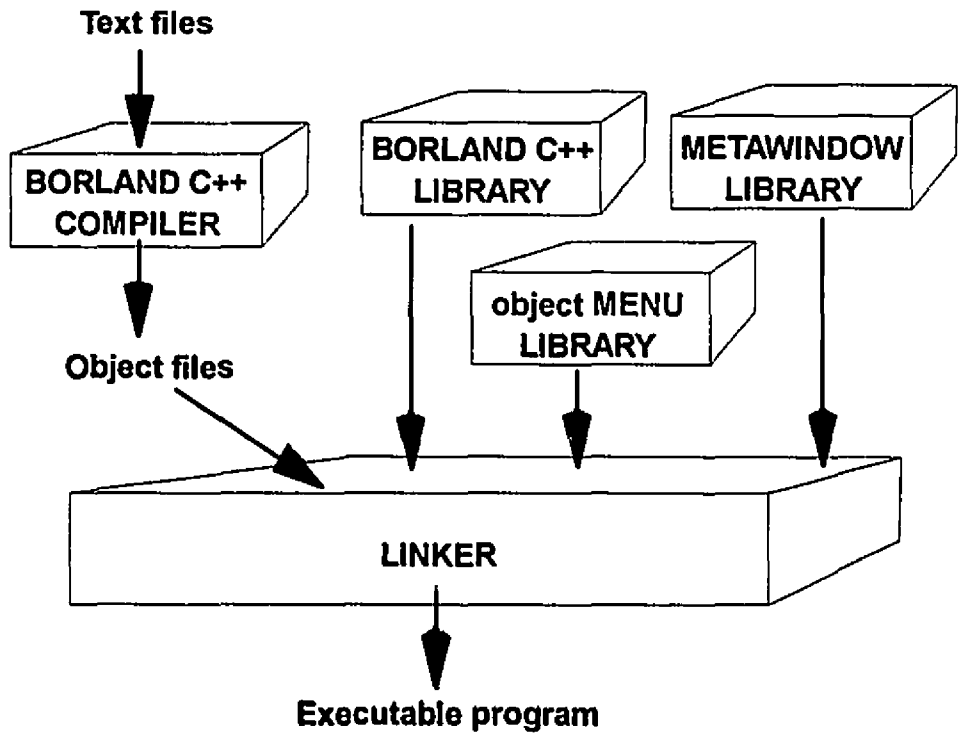

Figure 3.12 **Compilation and linkage processes**

As mentioned earlier, a makefile was used to build the executable program. This file is listed below:

```
#FILE: lmc.mak -- makefile to build lmcgui.exe
# Section 1
OBJS = m0.obj m1.obj m2.obj m3.obj m4.obj m5.obj m6.obj
m7.obj m8.obj
```

```
OBJS2 = m1.obj m2.obj m3.obj m4.obj m5.obj m6.obj m7.obj
m8.obj
DIR = g:\apps\borlandc\bin\\
HS = m0.h m1.h m2.h m3.h m4.h m5.h m6.h m7.h
CFLAGS=-f287 -k- -N -m1 -DMETAVER4=1 -c
# Section 2
all: lmcdsp1.LOD $(OBJS) m0.EXE
lmcdsp1.LOD : lmcdsp1.ASM
    c:\dsp56\bin\asm56000 -A -Blmcdsp1.LOD -OS,SO,CRE -L
lmcdsp1
m0.OBJ : m0.CPP $(HS)
    $(DIR)bcc $(CFLAGS) m0.cpp
m1.OBJ : m1.CPP $(HS)
    $(DIR)bcc $(CFLAGS) m1.cpp
m2.OBJ : m2.CPP $(HS)
    $(DIR)bcc $(CFLAGS) m2.cpp
m3.OBJ : m3.CPP $(HS)
    $(DIR)bcc $(CFLAGS) m3.cpp
m4.obj : m4.cpp $(HS)
    $(DIR)bcc $(CFLAGS) m4.cpp
m5.obj : m5.cpp $(HS)
    $(DIR)bcc $(CFLAGS) m5.cpp
m6.obj : m6.cpp $(HS)
    $(DIR)bcc $(CFLAGS) m6.cpp
m7.obj : m7.cpp $(HS)
    $(DIR)bcc $(CFLAGS) m7.cpp
m8.obj : m8.cpp $(HS)
    $(DIR)bcc $(CFLAGS) m8.cpp
m0.EXE : $(OBJS)
   bcc -f287 -k- -N -m1 -Y m0.obj -Yo $(OBJS2) m4.lib -Yo-
m2.lib mY.lib
```

The makefile is structured in two sections: section 1 gathers the macro definitions that will be used throughout this file and section 2 represents the sequence of compiling and linking of both the DSP software and the graphical user interface.

In section 1, OBJS and OBJS2 are defined and represent the object files for all the modules used for the graphical user interface. The only difference between OBJS and

OBJS2 is that OBJS2 does not contain the object file of the main (m0.obj) because in the linking stage one has to exclude the main. In the same section, the DIR macro definition contains the path where the system can find the compiler and the linker (with the name bcc.exe). Another macro definition, HS, contains the collection of the header files and finally, CFLAGS is a macro definition for all the options used to customize the compilation and linkage processes. The parameters used for the present compilation are:

-f287 this option tells the compiler to generate floating-point operations using inline 80287 (or higher) instructions rather than using calls to 80287 emulation library routines.

-k this option generates a standard stack frame, which is useful when using a debugger to trace back through the stack of called subroutines.

-N this option generates stack overflow logic at the entry of each function, which causes a stack overflow message to appear when a stack overflow is detected. This is costly in terms of both program size and speed but is provided as an option because stack overflows can be very difficult to detect.

-ml this option tells the compiler to compile using the large memory model. When a module is compiled, the resulting code for that module cannot be greater than 64 K, since it must all fit inside one code segment. Because the initial modules were too big to fit into one (64K) code segment, one must break them up into different source code files, compile each file separately, then link them together.

-DMETAVER4=1 this option lets the user define the current version of METAWINDOW (in this case, version 4 is set to true).

-c this option compiles and assembles the named .CPP and .ASM files, but does not execute a link command.

Section 2 starts with the line `all: lmcdsp1.LOD $(OBJS) m0.EXE` which represents all the targets we want the make utility to update and this appears in the command line `make -f lmc.mak all > lmc_err.txt`. This command line is used to execute the makefile lmc.mak with the targets specified in `all` and the output redirected to a file called `lmc_err.txt`, file used for debugging purposes. The files included in the `all` line consist of the compiled DSP driver ,the object files (m0 - m8), and the final result, the file m0.exe. The `lmcdsp1.LOD` file represents the outcome of the compilation of the file `lmcdsp1.ASM` using the DSP 56000 macro assembler. For details related to the DSP driver, please see [Shi 94].

The following lines of the makefile show the procedure for obtaining each object file from the respective source code text file. For example, the M0 object file is the target

file which is obtained from the `M0.CPP` file and the header files `M0.H-M7.H`. The makefile checks if any of these files is updated and, if this is the case, activates the command-line compiler with the selected options CFLAGS.

The last three lines show the target file as being `m0.exe`, built by linking the object files taking into account the specified options and combining them with the available libraries.

The executable `m0.exe` is the result of compilation and linkage of all the source code files and, in the next chapter, different screens of this software will be presented as results of the work.

Chapter 4: RESULTS AND CONCLUSIONS

The new LiMCA software was tested while doing many experiments for both water and aluminum. The degree of utilization of the DSP coprocessor board can give us an idea of the potential for future development such as implementing the Peak Classification Task at the DSP level and developing code to use DAC channels for process control. Figure 4.1 shows the degree of utilization mentioned above.

The calculation of the usage by all the processes in this figure was based upon the

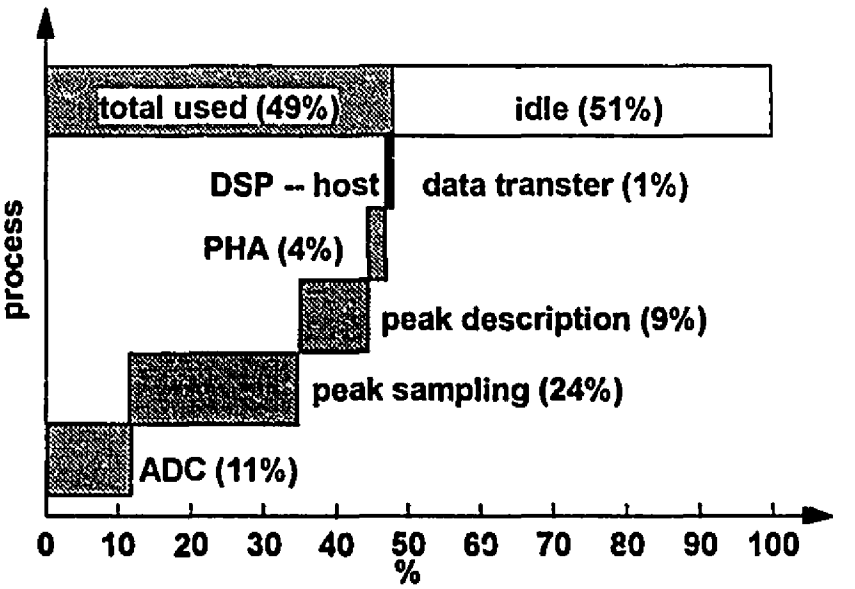

Figure 4.1 Usage of the DSP CPU

worst case which was mentioned in Section 2.2.1. The assumptions are that the DSP real-time software is working in the two-channel mode and that the ADC sampling rate is set to 50 kHz to avoid aliasing of the input signal. Considering these assumptions and the worst operating conditions, i.e. 2000 peaks per second, the DSP processor is used 49% of its capacity.

As for the host-DSP interface, the same experiments performed in water and aluminum showed that there were no detrimental delays introduced to the DSP process by the GUI. For the amount of data to be transferred from the DSP to the host, the interface has not reached its full capacity. The high efficiency of the interface is attributed to the

successful memory management and synchronization between the background and front functions.

The following figures are some of the screens of the graphical user interface, presented here in order to show how the user can navigate through this interface. Based on the tree structure presented in Section 3.4, one can follow the screens that are shown below and that were captured from the computer's monitor.

Figure 4.2 presents the "LiMCA Metal/Water Window" which is the opening window of the user interface. Inside this window, the "Principal horizontal menu" containing three items can be seen. The "On-Line-Acquire" item was selected and a vertical menu composed of two items, "Setup" and "Start", comes up. After "Setup" was chosen, the user can select between starting a "New test" or to "Continue test". In the case presented in Figure 4.2, a "New test" was selected and another vertical menu comes up containing two items: "Templates" and "Previous". In Figure 4.2, the "Previous" selection was made and another box containing test titles and locations comes up. This list enables the user to select one of the previous tests as the starting point in a "New test".

Figure 4.3 shows the same principal window but now the beginning of a "New test" is selected from the "Template" list. As one can notice, the user can choose between three available templates: "NEW", "Water", and "Aluminum". The "Water" and "Aluminum" templates contain commonly used parameters that the operator will see as

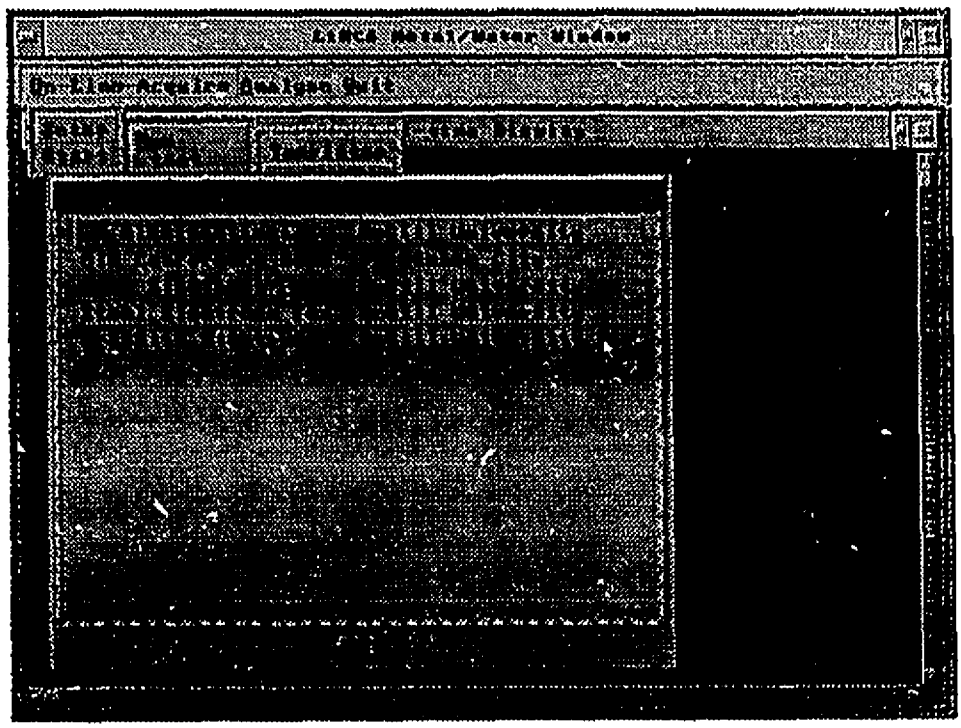

Figure 4.2 Using a previous test as starting point for a new one

soon as he chooses the respective template. The "NEW" item gives the possibility to save another template than the two specified. This new template is saved into memory and added in the "Template Choices" dialog box that can be seen in Figure 4.3.

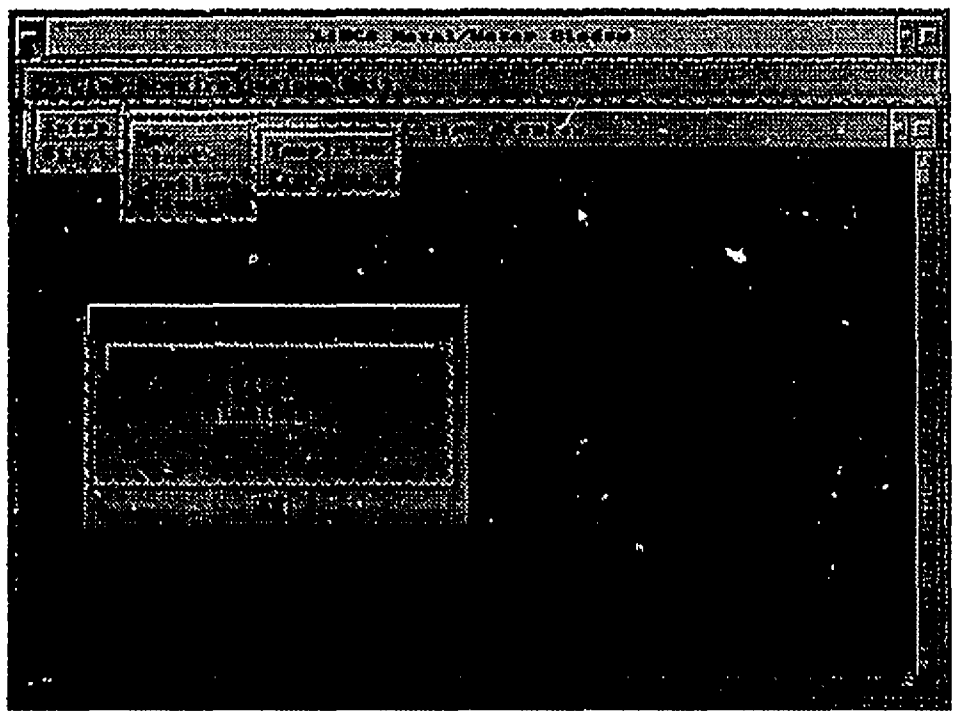

Figure 4.3 Using templates to start a new test

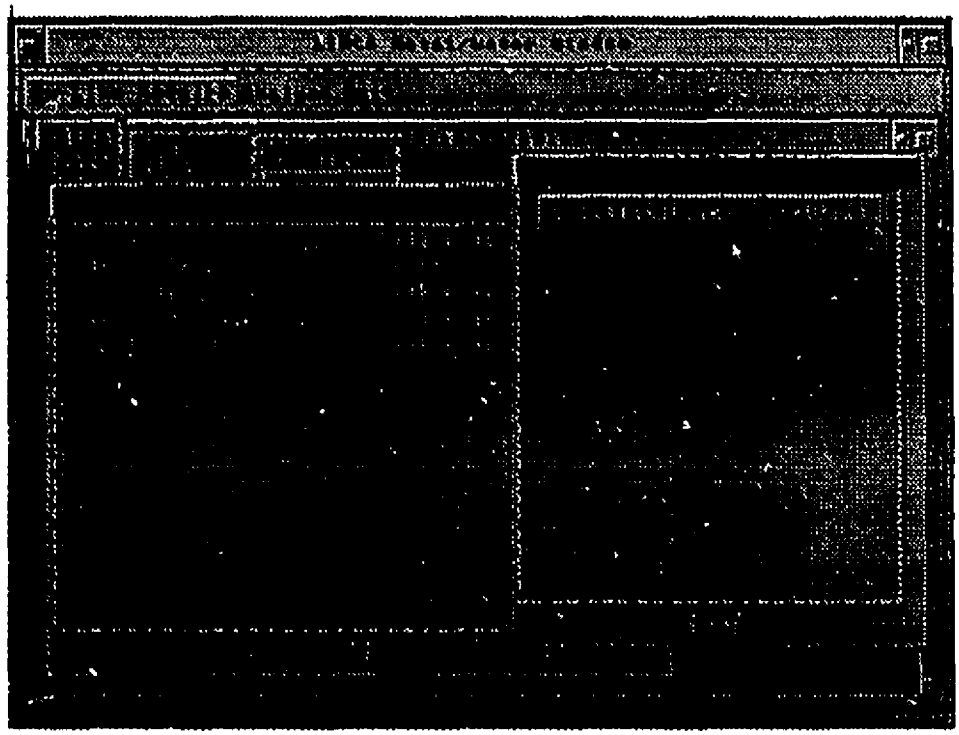

Figure 4.4 Continuing an acquisition

Another feature of the user interface is that the operator can browse previous tests and acquisitions as presented in Figure 4.4. The user can choose one of the previous tests, and a dialog box containing a list of acquisitions corresponding to the respective test comes on the screen. The operator does not have to input anything in the editing fields

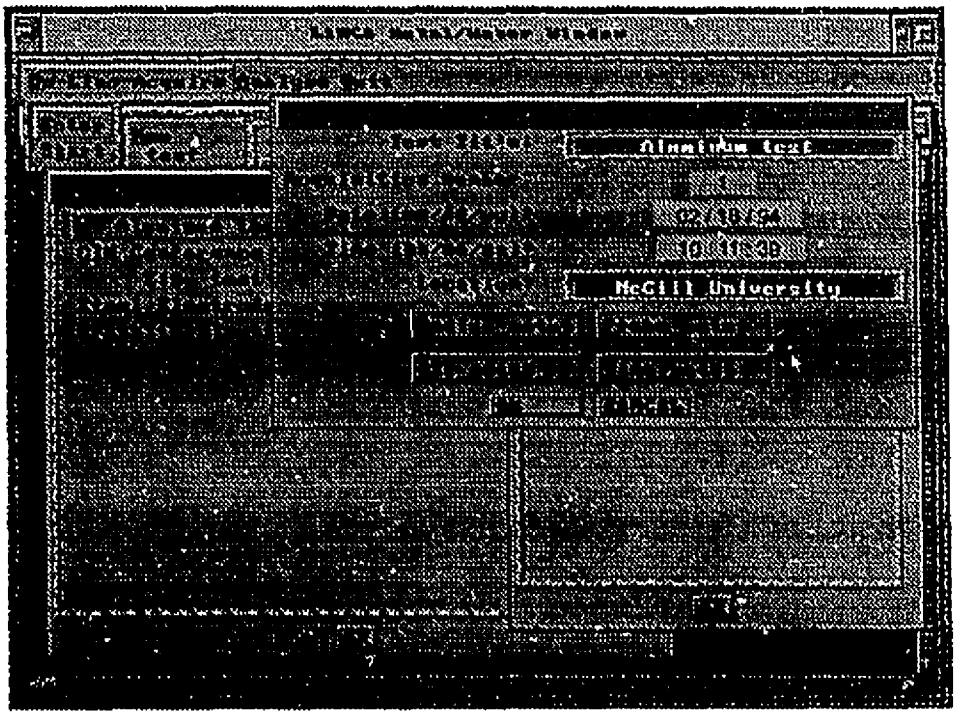

Figure 4.5 Initiating the "Setup" for a new test

because these fields are automatically updated and he/she can start a new acquisition using previous elements.

Figures 4.5 and 4.6 show a sample of the setup window. In Figure 4.5, the active dialog box is the "Setup Form" dialog box which contains general information (i.e. title, acquisition number, date, location, etc.) about the test that will be performed. The user is allowed to type different information about the experiment, information that is saved and will distinguish the respective test. There are also four buttons which enable access to the four "child" dialog boxes which are necessary for inputting or adjusting the different parameters of the experiment.

In Figure 4.6, one of the "child" dialog boxes of the "Setup" box, "Probe Setup" dialog box, is activated by clicking one of the four buttons inside the "Setup" dialog box. It contains different fields that can be edited by the operator, but also some of the values are directly edited by the software itself. These values are loaded in the software for some

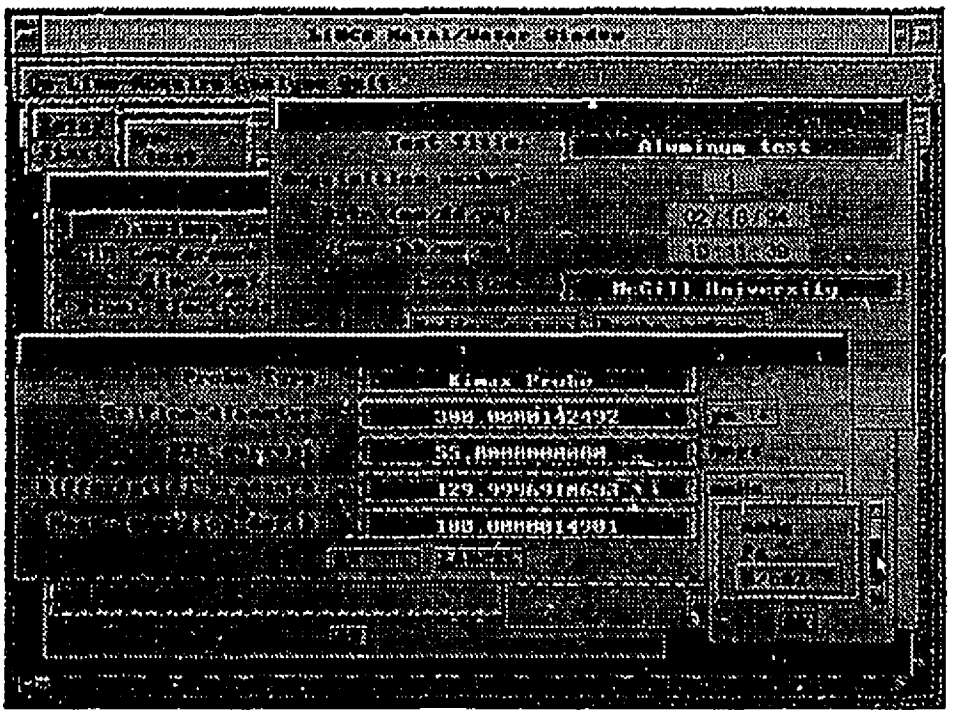

Figure 4.6 Activated "Probe Setup" dialog box

of the templates in use at this moment. Also, one can notice that for most of the input fields, unit conversion pop-up dialog boxes are attached. In each of these, the desired unit of measurement is selected using a scroll bar and the corresponding value is automatically calculated and displayed in the associated field. Several choices of units were given, standard SI as well as industry used units.

Figure 4.7 shows the calibration signal used to tune the LiMCA apparatus before any experiment is started. This calibration signal is displayed in the real-time display window which has a graphical display area and a data display area.

Figure 4.8 demonstrates the real-time window for an aluminum test. In the case presented, the inclusion counts per PHA channel are displayed in real-time. Finally, Figure 4.9 shows a sample of the analysis window. It appears also as two areas: one for graphical display, the other for data display. Several buttons and one line cursor are available for data analysis. One should mention that in Figures 4.8 and 4.9 different types of peaks are counted and displayed in different colors. These cannot be seen in the given representation. The counts for normal pulses are labeled with NP and the baseline jumps with BJ.

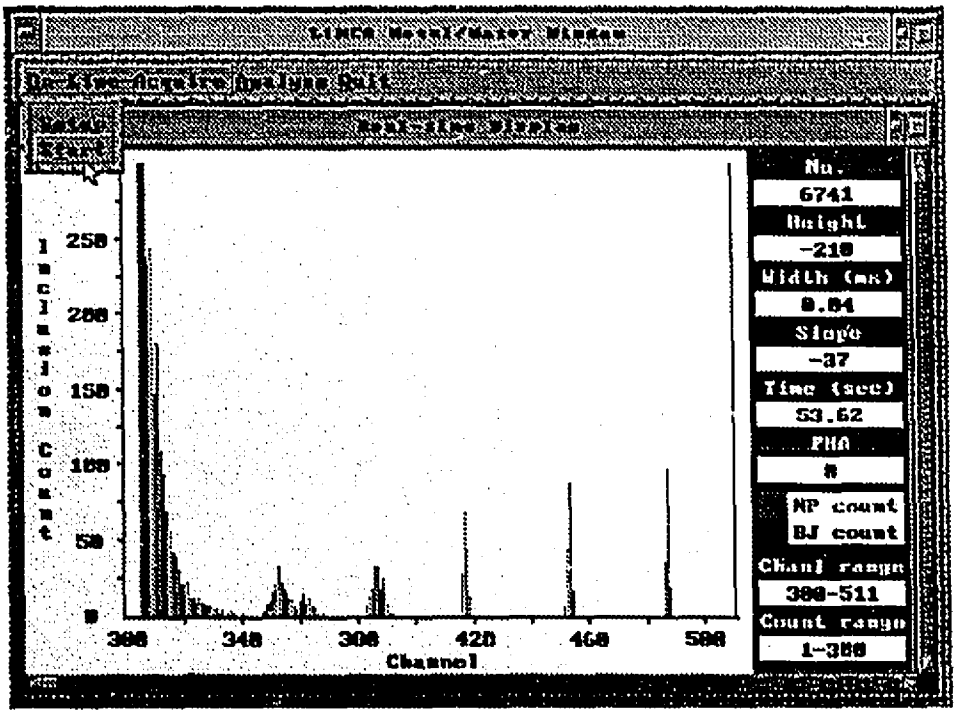

Figure 4.7 Real-time display - Calibration signal

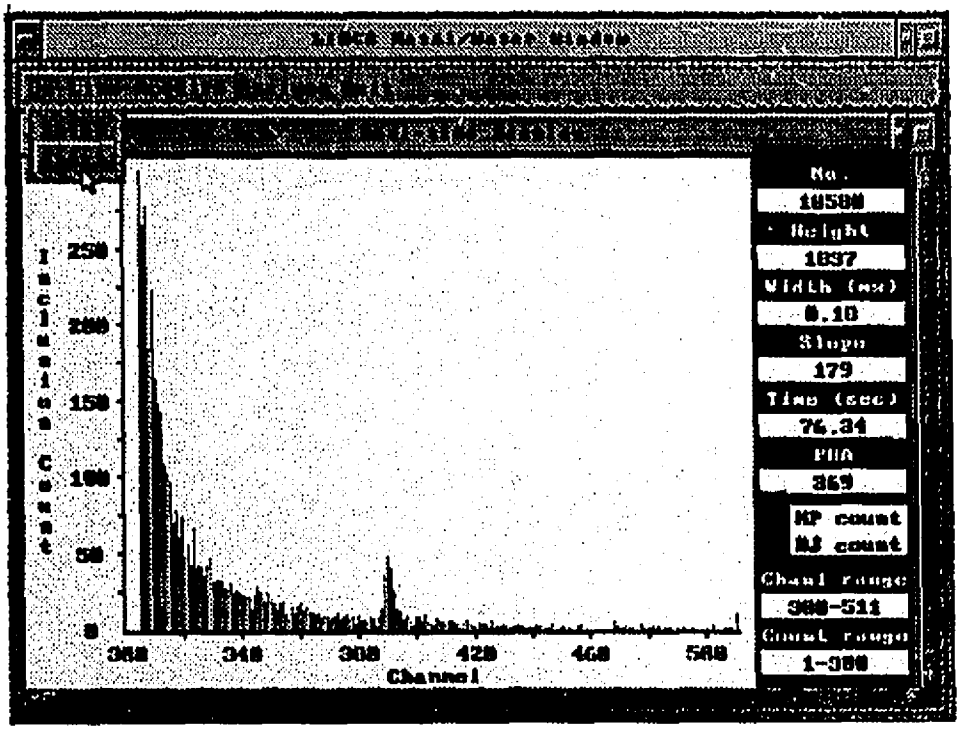

Figure 4.8 Real-time display window - Aluminum test

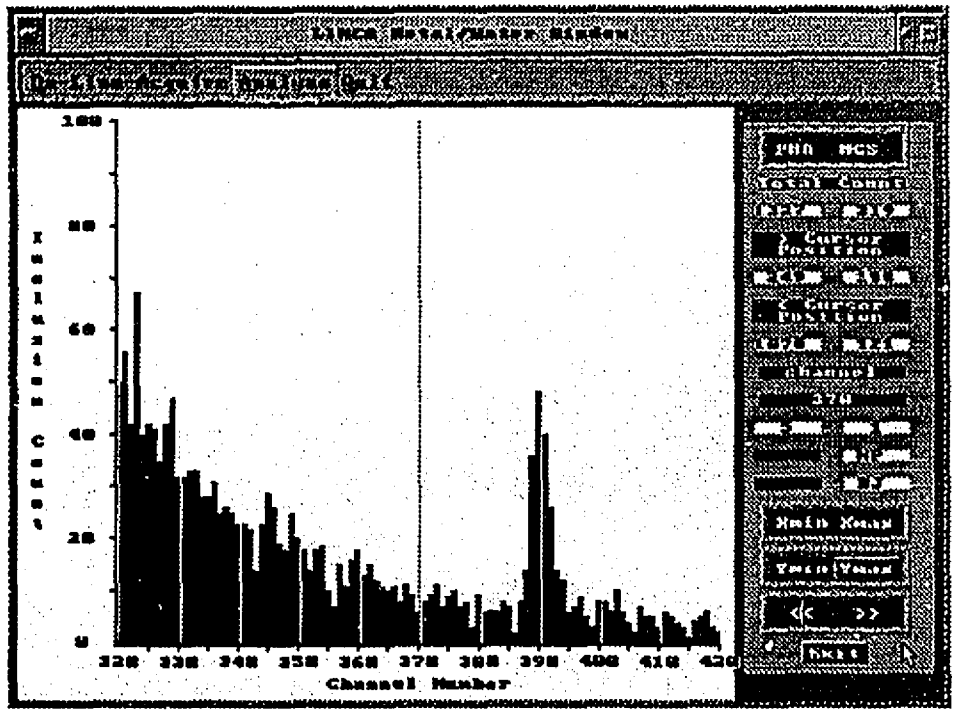

Figure 4.9 An example of the "Analysis" window

Conclusions

A DSP-based LiMCA system has been implemented to replace the first generation LiMCA system, which is built using the analog signal processing technique. The DSP real-time software, the host-DSP interface and the GUI have been implemented and tested. All these software parts are adequate to support the worst case of LiMCA operation. The application of DSP hardware and software provides a software-based signal processing system. The system is software controlled, cost effective, and easier to adapt to different situations.

Object-oriented programming is applied in the implementation of the graphical user interface. Because of the reusability of the code, class inheritance, and data encapsulation, this programming technique is very well suited to the system's development for which frequent modifications and improvements have to be done as the project is continuing.

This system allows us to continue the investigation into the possibility of discriminating the types of inclusions in liquid metals based on the Electric Sensing Zone principle. To enhance the performance of the DSP LiMCA system, some improvements are projected:

- The implementation of a sharp notch filter is needed to eliminate frequency components that interfere with the LiMCA signal. These components are due to the

electrical noises generated by other equipment used in an industrial environment where the experiments take place. The filter can be built upon a fast low-price DSP board and can communicate with the DSP-56 board through its network port.

• Although the peak description parameters are saved, the sampled LiMCA peaks also have to be saved for research purposes. Another DSP process has to be designed and implemented to transfer the sampled peak through the host-DSP interface into a fast hard drive using the SCSI (Small Computer System Interface) of the DSP-56 hardware.

• A compression algorithm and its implementation should be considered to decrease the size of the files containing the sampled peaks and the peak description parameters.

• A table-driven peak classification algorithm can be implemented according to the characteristics of the peaks given by the peak description parameters. Further studies have to be conducted concerning the peak classification algorithm, especially on the classification of the Multiple Pulses. Finally, the classification algorithm has to be embedded in the DSP software. This part of the future work is important because the peak classification makes possible the elimination of false counts and the automation of the orifice conditioning.

• A software LiMCA signal simulator is needed to study the high pass filter effect and to compensate the magnitude attenuation of the LiMCA peaks.

REFERENCES

[Ariel 89] Ariel Corporation, *User's Manual for the DSP-56 DSP Coprocessor Board for PC Compatibles*, Ariel Corporation, 1989

[Bates and Hutter 81] D.A. Bates and L.C.Cutter, "An Evaluation of Aluminium Filtering Systems using a Vacuum Filtration Sampling Device", *Light Metals*, The Metallurgical Society of AIME, pp. 707-721, 1981.

[Bauxman et al. 76] K. Bauxman, J.D. Bornand, G.B. Leconte, "Impact of Purification Methods on Inclusions and Melt Loss", *Light Metals*, The Metallurgical society of AIME, pp. 191-207, 1976.

[Booch 91] G. Booch, "Object Oriented Design With Applications", The Benjamin/Cummings Publishing Company, 1991.

[Carayannis et al. 92] G. Carayannis, F. Dallaire, X. Shi, R.I.L. Guthrie, "Towards Intelligent Detection of Inclusions in Liquid Metals", *Proc. Int. Symposium on Artificial Intelligence in Materials Processing Operations*, 31st CIM Conf. of Metallurgists, Edmonton (Alberta), pp. 227-244, Aug. 1992.

[Coulter 56] W. H. Coulter, "High Speed Automatic Blood Cell Counter and Cell Size Analyzer", *Proc. of the National Electronic Conf.*, pp. 1034-1042, Chicago (IL), 1956.

[Dallaire 90] F. Dallaire, "Electric Sensing Zone Signal Behaviour in Liquid Aluminum", *Master's Thesis*, Dept. of Mining & Metallurgical Eng., McGill University, 1990.

[DeBlois and Bean 70] R.W. DeBlois, C.P. Bean, "Counting and Sizing Submicron Particles by the Resistive Pulse Technique", *The Review of Scientific Instruments*, Vol. 41, No. 7, pp. 909-915, 1970.

[Doutre 84] D.A. Doutre, "The Development and Application of a Rapid Method of Evaluating Molten Metal Cleanliness", *Ph.D. Thesis*, Dept. of Mining & Metallurgical Eng., McGill University, 1984.

[Foley et al. 90] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, "Computer Graphics, Principles and Practice", Addison-Wesley Publishing Company, 1990.

[Kulunk 92] B. Kulunk, "Kinetics of Removal of Calcium and Sodium by Chlorination from Aluminum and Aluminum-1WT% Magnesium Alloys", *Ph.D Thesis*, Dept. of Mining & Metallurgical Eng., McGill University, 1992.

[Kuyucak 89] S. Kuyucak, "On the Direct Measurement of Inclusions in Molten Metals", *Ph.D. Thesis*, Dept. of Mining & Metallurgical Eng., McGill University, 1989.

[Kuyucak and Guthrie 89] S. Kuyucak, R.I.L. Guthrie, "On the Measurement of Inclusions in Copper-Based Melts", *Can. Met. Quart.*, Vol. 27, pp. 41-48, 1989.

[Laurel 90] B. Laurel, "The Art of Human-Computer Interface Design", Addison-Wesley Publishing Company, 1990.

[Lee 91] H.C. Lee, "On the Development of a Batch Type Inclusion Sensor in Liquid Steel", *Ph.D. Thesis*, Dept. of Mining and Metallurgical Eng., McGill University, 1991.

[Levy 81] S.A. Levy, "Applications of the Union Carbide Particulate Tester", *Light Metals*, The Metallurgical Society of AIME, pp. 723-733, 1981.

[Mansfield 82] T.L. Mansfield, "Ultrasonic Technology for Measuring Molten Aluminium Quality", *Light Metals*, The Metallurgical Society of AIME, pp. 969-980, 1982.

[Motorola 89] Motorola, *DSP56000/DSP56001 Digital Signal Processor User's Manual*, 1989

[Motorola 92] Motorola, ``24-Bit General Purpose Digital Signal Processor", *Motorola Semiconductor Technical Data*, Rev.3, 1992

[object-Menu 92] Island Systems, *object-Menu - The professional Graphical User Interface toolkit for C++*, Island Systems, 1992.

[Oppenheim and Schafer 89] A.V. Oppenheim, R.W. Schafer, *Discrete-time Signal Processing*, Prentice-Hall, 1989.

[Pitcher and Young 69] D.E. Pitcher, "Methods of an Apparatus for Testing Molten Metal", *U.S. Patent*, 3,444,726, May 20, 1969.

[Shi 94] X. Shi, "Upgrading Liquid Metal Cleanliness Analyzer (LiMCA) with Digital Signal Processing (DSP) Technology", *M.Eng. Thesis*, Dept. of Mining & Metallurgical Eng., McGill University, 1994.

[Shneiderman 92] B. Shneiderman, "Designing the User Interface, Strategies for Effective Human-Computer Interaction", 2nd Edition, Addison-Wesley Publishing Company, 1992.

[Siemensen 81] C.J. Siemensen, "Sedimentation Analysis of Inclusions in Aluminium and Magnesium", *Met. Trans. B.*, Vol. 12B, pp. 733-743, 1981.

[Thibault et al. 89] J.-F. Thibault, A. Boisset, F. Dallaire, G. Carayannis, "Pattern Recognition Techniques for Metal Quality Control", *Canadian Conf. on Electrical and Computer Engineering*, Montréal (Québec), pp. 771-774, September 1989.

[Thimbleby 90] H. Thimbleby, "User Interface Design", Addison-Wesley Publishing Company, 1990.

[Tian et al. 92] C. Tian, F. Dallaire, R.I.L. Guthrie, "Inclusion Removal from Aluminum Melts through Filtration", *Proc. Advances in Production and Fabrication of Light Metals and Metal Matrix Composites*, 31st CIM Conf. of Metallurgists, Edmonton (Alberta), pp. 153-161, Aug. 1992.

[Yamanoglu 92] G. Yamanoglu, "Characterization of Submerged Powder Injection into Water Using an In-lining Particle Detection System", *Master's Thesis*, Dept. of Mining & Metallurgical Eng., McGill University, 1992.

APPENDIX A: DECODING PEAK PARAMETERS FROM DSP FORMAT INTO HOST FORMAT

M6.CPP

```
//-----------------------------------------------------
//convert peak data from DSP format to HOST format

void PkParmConvert( PK_STRUCT *orig_pk, PK_STRUCT1 *peak)
     {
   unsigned long base_count;
   base_count=(unsigned long)orig_pk->start_time_hi*65536L
              +(unsigned long)orig_pk->start_time_lo;
   peak->start_time=(float)base_count/WorkingSetup.SFreq;
   peak->start_slope=orig_pk->start_slope;
   peak->max_time=(float)(base_count+(unsigned
     long)orig_pk->max_time)/ WorkingSetup.SFreq;
    peak->height=orig_pk->height;
    peak->width=(float)orig_pk->end_time*1000.0 /
     WorkingSetup.SFreq;
    peak->end_slope=orig_pk->end_slope;
    peak->cha_num=orig_pk->cha_num;
    }
```

M6.H

```
typedef struct PK_PARM { //data structure for raw peak
                         //parameters
    unsigned start_time_hi;   //high 16 bit of start time
    unsigned start_time_lo;   //low 16 bit of start time
    int start_slope;
    int max_time;             //time at max
    int height;
    int end_time;             //time at end
```

```
    int end_slope;
    int cha_num;    //PHA channel number, + for channel A,
                //- for channel B
    } PK_STRUCT, *PK_STRUCT_PTR;

typedef struct PK_PARM1  {    //data structure for
                              //converted parameters
    float start_time;    //start time in second
    int start_slope;     //identical to the one in PK_PARM
                    //struct
    float max_time;      //time at max in second
    int height;          //identical to the one in PK_PARM
                    //struct
    float width;         //peak width in ms
    int end_slope;       //identical to the one in PK_PARM
                    //struct
    int cha_num;    //identical to the one in PK_PARM
                //struct
    } PK_STRUCT1, *PK_STRUCT1_PTR;
```

APPENDIX B: LIST OF MODULES FOR THE LiMCA GRAPHICAL USER INTERFACE

M0.H	Header file for unit definitions.
M1.H	Header file for unit structure.
M2.H	Header file for LiMCA setup.
M3.H	Class definitions header file for our own derived classes.
M4.H	The prototypes of the LiMCA setup functions.
M5.H	Class definitions header file for the setup parameters and method functions.
M6.H	Header file for different parameters definitions.
M7.H	Header file used to integrate MetaWINDOW and object-Menu.

M0.CPP	Main program.
M1.CPP	Class functions.
M2.CPP	Functions for LiMCA setup.
M3.CPP	Class function body of the DSP loader.
M4.CPP	Global functions for the DSP loader.
M5.CPP	Low-level interface functions for the DSP process.
M6.CPP	Real-time functions.
M7.CPP	Functions for the EMS management.
M8.CPP	Functions for LiMCA setup.

APPENDIX C: THE CONSTRUCTOR AND INTERFACE FUNCTIONS FOR THE CLASS SetupFormDialog

M1.CPP:

```
//m1.cpp: body of the class SetupFormDialog
//------------------- for Class SetupFormDialog -------

SetupFormDialog::SetupFormDialog( void ):omDialog (0,0)    {

    int label_font = 9;
    int input_font = 9;
    int input_color = YELLOW;
    int input_back_color = CYAN;
    omDressUpTypes input_dress = RIDGE;
    int menu_font = 9;
    int input_str_len = 24;
    int label_str_len = 20;
    int title_color =RED;
    int title_font = 20;
    char *init_str = "111111111111111111111111";
    char *pic_str = "                              ";
    char s[25];

//-------some general info input fields----------------

//Field 1: test title
    strcpy( s, "Test Title: ");
    TextLengthInc( s, label_str_len, ALIGN_RIGHT );
    l1  = new omLabel( s );
    l1 -> usedNew = TRUE;
    l1 -> menuFont.omSetFont( label_font );

    te1 = new myTextEditor( init_str );
    te1  -> usedNew = TRUE;
```

```
    te1 -> inputFont = input_font;
    te1 -> inputFontColor = input_color;
    te1 -> dress = input_dress;
    te1 -> setPicture( pic_str );

//Field 2: acquisition number
    strcpy( s, "Acquisition number: ");
    TextLengthInc( s, label_str_len, ALIGN_RIGHT );
    l2  = new omLabel( s );
    l2 -> usedNew = TRUE;
    l2 -> menuFont.omSetFont( label_font );

    itoa(WorkingSetup.AcqNo, s, 10);
    TextLengthInc( s, input_str_len, ALIGN_CENTER );
    l2a = (omLabel *) new omLabel( s );
    l2a -> usedNew = TRUE;
    l2a -> menuFont.omSetFont( input_font, 1, input_color );
    l2a -> bkColor = input_back_color;

//Field 3: date
   // 'Date (mm/dd/yy): ' will be a label
               //the current date is always
    strcpy(s, "Date (mm/dd/yy): ");
    TextLengthInc( s, label_str_len, ALIGN_RIGHT );
    l3  = new omLabel( s );
    l3 -> usedNew = TRUE;
    l3 -> menuFont.omSetFont( label_font );

    _strdate( s);
    TextLengthInc( s, input_str_len, ALIGN_CENTER );
    l3a = (omLabel *) new omLabel( s );
    l3a -> usedNew = TRUE;
    l3a -> menuFont.omSetFont( input_font, 1, input_color );
    l3a -> bkColor = input_back_color;

//Field 4: time
```

```
    strcpy(s, "Time (hh/mm/ss): " );
    TextLengthInc( s, label_str_len, ALIGN_RIGHT );
    l4  = new omLabel( s );
    l4 -> usedNew = TRUE;
    l4 -> menuFont.omSetFont( label_font );

    _strtime(s);
    TextLengthInc( s, input_str_len, ALIGN_CENTER );
    l4a = (omLabel *) new omLabel( s );
    l4a -> usedNew = TRUE;
    l4a -> menuFont.omSetFont( input_font, 1, input_color );
    l4a -> bkColor = input_back_color;

//Field 5: test location
    strcpy(s, "Location: ");
    TextLengthInc( s, label_str_len, ALIGN_RIGHT );
    l5  = new omLabel( s );
    l5 -> usedNew = TRUE;
    l5 -> menuFont.omSetFont( label_font );

    te5 = new myTextEditor( init_str );
    te5  -> usedNew = TRUE;
    te5 -> inputFont = input_font;
    te5 -> inputFontColor = input_color;
    te5 -> dress = input_dress;
    te5 -> setPicture( pic_str );

//--------------- Medium dialog ------------------

    md = (myMetalDialog* ) new myMetalDialog;
    md -> usedNew = TRUE;
    md -> setTitleBkColor (input_back_color);
    md -> setTitleFont( menu_font, title_color, title_font);

//------------------ probe dialog ------------
```

```
    pd = (myProbeDialog* ) new myProbeDialog;
    pd -> usedNew = TRUE;
    pd -> setTitleBkColor (input_back_color);
    pd -> setTitleFont( menu_font, title_color, title_font);

//--------------------DSP dialog ----------------

    dd = new myDSPDialog;
    dd -> usedNew = TRUE;
    dd -> setTitleBkColor (input_back_color);
    dd -> setTitleFont( menu_font, title_color, title_font);
//------------------ Time dialog ---------------

    td = new myTimeDialog;
    td -> usedNew = TRUE;
    td -> setTitleBkColor (input_back_color);
    td -> setTitleFont( menu_font, title_color, title_font);

//-------- Button Menu to access the above four dialogs---

    bm = (omButtonMenu*) new omButtonMenu( 4,     //total
buttons
                0,0,     // upper left corner
                0,0,     // no menu hot-key
                2,2);    // 2 rows x 2 cols

    bm -> usedNew = TRUE;
    bm -> setMenuFont(menu_font, 1, input_color);

    *bm + LITERAL_HOTKEYS
     + "~Medium_setup" + LU1(PassXY2dMedium, md) + *md
     + "~Probe_setup"  + LU2(PassXY2dProbe,  pd) + *pd
     + "~DSP_setup"    + LU3(PassXY2dDSP,    dd) + *dd
     + "~Time_setup"   + LU4(PassXY2dTime,   td) + *td;
```

```
    bm -> setHotkey(0, K2_F2);      //sets F2 as a hotkey for
this button menu

  /**** this is OK/Cancel buttons inside Real-time Display
******/
 ok = ( omButtonMenu* ) new omButtonMenu ( 2, 0,0,0,0,1,2 );
    *ok + LITERAL_HOTKEYS
     + "~OK"
     + "~CANCEL" + ResumedSetupForm
     + AUTO_BUTTONUP + IS_RADIO + IS_GROUP;

    ok -> m[0].exitType = omMODAL_EXIT;
    ok -> eventType = omGROUP;

    *this + WITH_TITLE("Setup")
       + *l1   + *te1
       - *l2   + *l2a
       - *l3   + *l3a
       - *l4   + *l4a
       - *l5   + *te5
       - *bm   + ALIGN_BOTTOM + ALIGN_MIDDLEH
       - *ok   + ALIGN_BOTTOM + ALIGN_MIDDLEH + IS_MODAL;
    }

omBoolean SetupFormDialog::doModalExit( void )     {

    int i;
    char s[26],s1[12],num[10];
//field 1: test title
    if( Status.NewTest || Status.NewTemplate )          {
     strcpy(WorkingSetup.TestName,  dSetupForm  ->  te1  ->
getField( 1 ) );

//field 5: test location
     strcpy( WorkingSetup.Location,  dSetupForm  ->  te5  ->
getField( 1 ) );
```

```
    }

   if( Status.NewTemplate )  {
    i = omReply( "Do you want to save this template?",
                  WorkingSetup.TestName, -1,
                  "~Yes", "~No");
    if( !i )  {
      AppendTemplateIndx( );

      dTemplates ->vms -> enableItem( TotalTemplates - 1 );
      strcpy(s, SetupTemplate[ TotalTemplates- 1].Name );
      TextLengthInc( s, 24, ALIGN_CENTER );
      dTemplates -> vms -> newText( TotalTemplates -1, s);
        }
    else {
        TotalTemplates--;
        }
    }
   else if( Status.NewTest ) {
    i = omReply( "Do you want to save this setup?",
                  WorkingSetup.TestName, -1,
                  "~Yes", "~No");
    if( !i )  {
                 AppendTestIndx(   WorkingSetup.TestName,
WorkingSetup.Location );
        AppendAcqIndx( "NULL" );
           SaveSetup( Tests[ ActiveTest ].TestDir, Acqs[
ActiveAcq ].SetupFile,
         SETUP_ID, ActiveTest, ActiveAcq );
        Status.NewTestSaved = TRUE;
        }
    }
   else  {
    i = omReply( "Do you want to save this setup?",
                  WorkingSetup.TestName, -1,
                  "~Yes", "~No");
```

```
    if( !i )  {
        if( Acqs[ ActiveAcq ].AcqFileFlag ) {
         AppendAcqIndx( "NULL" );
         dPrevious -> vmsq -> enableItem(  ActiveAcq );

         strcpy(s, "Acquisition" );
         itoa( Acqs[ ActiveAcq ].AcqNo, num, 10 );
         strcat ( s, " ");
         strcat(s, num);
         TextLengthInc( s, 15, ALIGN_CENTER );
         strcpy( s1, Acqs[ ActiveAcq ].Date );
         TextLengthInc( s1, 10, ALIGN_CENTER );
         strcat( s, s1);
         dPrevious -> vmsq -> newText( ActiveAcq, s);
         }

           SaveSetup(  Tests[  ActiveTest  ].TestDir,  Acqs[
ActiveAcq ].SetupFile,
               SETUP_ID, ActiveTest, ActiveAcq );
        }
     }
    Status.NewTemplate = FALSE;
    Status.NewTest = FALSE;
    return FALSE;
    }
```

APPENDIX D: LINE EDITOR CLASSES

M3.H:

```
//FILE: M3.H

//class definitions for our own derived class
//for lmc GUI

//-------------- CLASS myDecimalEditor ---------------
class myDecimalEditor: public omLineEditor
    {
    public:

       myDecimalEditor( char *any_line, int maxlen=-1 ):
       omLineEditor( any_line, maxlen ) {};  // constructor

       omBoolean interceptFunction();

     };

//-------------- CLASS myTextEditor ------------------
class myTextEditor: public omLineEditor
    {
    public:

       myTextEditor( char *any_line, int maxlen=-1 ):
       omLineEditor( any_line, maxlen ) {};  // constructor

       omBoolean interceptFunction();

     };
```

```
//----------------- CLASS myIntEditor -----------------
class myIntEditor: public omLineEditor
 {
  public:

   myIntEditor( char *any_line, int maxlen=-1 ):
   omLineEditor( any_line, maxlen ) {}; // constructor

   omBoolean interceptFunction();
 };
```

from m1.cpp

```
//---------- for class myDecimalEditor -------------
omBoolean myDecimalEditor::interceptFunction( void )    {

    int i,j=0;
    char *dot_ptr;
    char s[25];

    if (omU.anyKbd)    {       // if keyboard hit...

     if (omU.ch1 == 13)   {           //intercept "Enter".
         i = 0;
         while(theString[i] != '\0')    {
          if(theString[i] > 48) j++;
          i++;
          }

         if (j == 0)   {
          omDisplayMessage("   Input   |   error !   ",
                     2500,    //delay
                      500, 285, // x & y
                     BLACK,     //textcolor
                     YELLOW);      //backcolor
          return FALSE;
```

```
          }
         if( (strchr( theString, '+') != NULL ) ||
             (strchr( theString, '-') != NULL ) )      {
          omDisplayMessage("Input format error",0);
          return FALSE;
          }
         if( (dot_ptr = strchr( theString, '.' )) != NULL )
     {
          if( strchr( dot_ptr + 1, '.') != NULL)  {
              omDisplayMessage("Input format error",0);
              return FALSE;
              }
          }
         for(j=i=0; theString[i] !='\0'; i++)
          if( theString[i] != ' ' )    s[j++] =
theString[i];
         s[j] = '\0';
         TextLengthInc( s, 24, ALIGN_CENTER );
         setField( 1,s);
         return FALSE;
         }
     }
    return FALSE;
    }

//---------- for class myTextEditor ----------------
omBoolean myTextEditor::interceptFunction( void ) {

    char s[25];
    int i;

    if (omU.anyKbd)   {      // if keyboard hit...

     if (omU.ch1 == 13)  {          //intercept "Enter".
         strcpy( s, theString );
         StringWords( s );
```

```
        TextLengthInc( s, 24, ALIGN_CENTER );
        setField( 1,s);
        return FALSE;
        }
     }
    return FALSE;
    }

//------------- for class myIntEditor ------------------
omBoolean myIntEditor::interceptFunction()  {
    int i, j = 0;
    char s[25];

    if (omU.anyKbd) {
     if (omU.ch1 == 13)  {
        for(i=0; theString[i] !='\0'; i++)  {
         if( theString[i] != ' ' ) {
            s[j++] = theString[i];
            }
         }
        s[j] = '\0';
        TextLengthInc( s, 24, ALIGN_CENTER );
        setField( 1,s);
        return FALSE;
        }
     }
    return FALSE;
    }
```

APPENDIX E: CLASS DEFINITIONS FOR DIALOG BOXES

M3.H:

```
//FILE: M3.H

//class definitions for our own derived class
//for lmc UGI

//-------------Class myMetalDialog ----------------------
class myMetalDialog:public omDialog  {

    public:
     omLabel *l1, *l2, *l3, *l4, *l4a, *l5;
     omVertMenuScroll *vms1, *vms2, *vms3;
     omScrollMenuDlg *smd1, *smd2, *smd3;
     omComboBox *cb1, *cb2, *cb3;
     myTextEditor *te1;
     myDecimalEditor *de2, *de3, *de4, *de5;
     omButtonMenu *ok;
     struct Value v2,v3,v4,v5;
    //constructor
    myMetalDialog( void );
    omBoolean doModalExit( void );
    };

//-----------Class myProbeDialog --------------------
class myProbeDialog:public omDialog  {

    public:
     omLabel *l1, *l2, *l3, *l3a, *l4, *l5;
     omVertMenuScroll *vms2, *vms4, *vms5;
     omScrollMenuDlg *smd2, *smd4, *smd5;
     omComboBox *cb2, *cb4, *cb5;
     myTextEditor *te1;
```

```
    myDecimalEditor *de2, *de3, *de4, *de5;
    omButtonMenu *ok;
    struct Value v2,v3,v4,v5;
   //constructor
   myProbeDialog( void );
   omBoolean doModalExit( void );
   };

//------------Class myDSPDialog ---------------------
class myDSPDialog:public omDialog  {

   public:
    omLabel *l1, *l1a, *l2, *l2a, *l3, *l3a, *l4, *l4a;
    omVertMenuScroll *vms4;
    omScrollMenuDlg *smd4;
    omComboBox *cb4;
    int n2, n3, n4;
    omButtonMenu *ok, *bt2, *bt3;
   //constructor
   myDSPDialog( void );
   omBoolean doModalExit( void );
   };

//-----------Class myTimeDialog ----------------------
class myTimeDialog:public omDialog  {

   public:
    omHorizMenu *hm1;
    omLabel *l2, *l3, *l4;
    myDecimalEditor *de2, *de3;
    myIntEditor *ie4;
    omVertMenuScroll *vms2, *vms3;
    omScrollMenuDlg *smd2, *smd3;
    omComboBox *cb2, *cb3;
    int n1, n4;
```

```
 struct Value v2,v3;
 omButtonMenu *ok;
//constructor
myTimeDialog( void );
omBoolean doModalExit( void );
};
```

APPENDIX F: SOURCE CODE LISTING OF THE MAIN PROGRAM

```
//M0.CPP

#include "om.h"
#include "metaver3.h"          //meta fonts

#include "m0.h"
#include "m1.h"
#include "m3.h"
#include "m2.h"
#include "m5.h"
#include "m4.h"
#include "m6.h"
#include "m7.h"

#define MAX_METAL_MEDIUMS 15

char *TestIndxFile = "testindx.ind";
int LoadTestInfo = FALSE;
int LoadAcqInfo = FALSE;

#define MAX_TESTS    20    //maximum number of test
directories

struct TestInfo Tests[MAX_TESTS];

int TotalTests = 0; //total number of tests
int ActiveTest = 0; //the active test of the tests

#define MAX_ACQS     30

char* AcqIndBase = "Acqs";
```

```
struct AcqInfo Acqs[MAX_ACQS];

int TotalAcqs = 0;  //total number of acquisitions
int ActiveAcq = 0;  //the active acquisition

//#ifdef NETDRIVE
    char* WorkingDir ="G:\\GROUPS\\LIMCA\\NEWDRIVE";
//#else
//    char* WorkingDir ="C:\\TEST";
//#endif

char *TemplateIndxFile = "template.ind";
Setup WorkingSetup;
struct MetalMediumStruct MetalMedium[ MAX_METAL_MEDIUMS ];
char *MetalMediumList = "mmedium.lst";
int TotalMetalMediums;
int TotalTemplates;

SetupFormDialog *dSetupForm;
SetupTemplateDlg *dTemplates;
SetupPreviousDlg *dPrevious;
AnalyseDlg *Adialog;
omRect *MessageBox;

struct SetupTemplateStruct SetupTemplate[ MAX_TEMPLATES + 1
];
struct StatusStruct Status;

int far *pha_npcount, *pha_bjcount;
char far *pha_cha_st;
int far *mcs_npcount, *mcs_bjcount;
//global flags
int DSPLoad = FALSE;
```

```
//-------------------------------------------------------
//-------------- MAIN -----------------------------------
//-------------------------------------------------------

extern unsigned _stklen=40000; // increase stack
extern unsigned _ovrbuffer = 4800; // 75K overlay buffer

#include "owmain.h"           //substitute for "main" or
"WinMain"
#ifndef DOSX286
    _OvrInitEms(0,0,16);// For Borland C++, no effect for
Microsoft C++
#endif
    EMS_init( );
    unsigned long size_int = sizeof( int ) * CHAN_NUM;
    unsigned long size_char = sizeof( char )* CHAN_NUM;

    if((pha_npcount=(int*)farmalloc(size_int))==NULL) {
     printf("memory error\n");
     exit(0);
     }
    if((pha_bjcount=(int*)farmalloc(size_int))==NULL) {
     printf("memory error\n");
     exit(0);
     }
    if((pha_cha_st=(char*)farmalloc(size_char))==NULL) {
     printf("memory error\n");
     exit(0);
     }
     if((mcs_npcount=(int*)farmalloc(size_int))==NULL) {
      printf("memory error\n");
      exit(0);
      }
     if((mcs_bjcount=(int*)farmalloc(size_int))==NULL) {
      printf("memory error\n");
      exit(0);
```

```
      }

    int i;
    for(i=0;i<CHAN_NUM;i++)
     pha_cha_st[i] = (char) pha_npcount[i] = pha_bjcount[i]
= 0;

    omPrefs.winMainMenu = FALSE;
    omPrefs.scrollDressType = BEVELOUT;
    strcpy(omFontTable[1].fileName, "extsys16.fnt");
    strcpy(omFontTable[2].fileName, "extsys24.fnt");

    int back_color = CYAN;
    int title_font = 20 ;
    int title_color = RED;
    int font_number = 10;
    int medium_type;
    int menu_font = 9;

//initialize status flages
    Status.NewTest = FALSE;
    Status.NewTemplate = FALSE;          //default status is
continuous test
    Status.NewTestSaved = FALSE;
    Status.NoData = TRUE;
    Status.SetupDone = FALSE;

    InitScratchSetup( &WorkingSetup, METAL );

    LoadTestInfo = LoadTestIndx();

    MessageBox = new omRect;
    dSetupForm = new SetupFormDialog;
    dSetupForm -> usedNew = TRUE;
    dSetupForm -> setTitleBkColor (back_color);
```

```
    dSetupForm -> setTitleFont( font_number, title_color,
title_font);

    dTemplates = new SetupTemplateDlg( dSetupForm );
    dTemplates -> usedNew = TRUE;
    dTemplates -> setTitleBkColor (back_color);
    dTemplates -> setTitleFont( font_number, title_color,
title_font);

    dPrevious = new SetupPreviousDlg( dSetupForm );
    dPrevious -> usedNew = TRUE;
    dPrevious -> setTitleBkColor (back_color);
    dPrevious -> setTitleFont( font_number, title_color,
title_font);

//************************************************************
*
//the vertical menu from New test
    omVertMenu vmNew(2);
    vmNew.setMenuFont(menu_font, 1, BLUE);
     vmNew + LITERAL_HOTKEYS
           + "~Templates" + LU1(PassXY2dTemp, dTemplates)
+ *dTemplates
           + "~Previous"  + LU2(PassXY2dPrevious, dPrevious)
+ *dPrevious;

//*****************************************************
//the vertical menu from "Setup"
//*****************************************************
    omVertMenu vmSetup(2);
    vmSetup.setMenuFont(menu_font, 1, BLUE);
     vmSetup + LITERAL_HOTKEYS
          + "~New| test"       + NewTestFlag + vmNew
          + "~Continue|  test" + NewTestFlag
          + LU3(PassXY2dPrevious, dPrevious) + *dPrevious;
```

```
//*****************************************************
// the principal window follows:
//*****************************************************
   omRect winRect (0, 0, 640, 480);
   omWindow w(winRect, omNO_PARENT, "LiMCA Metal/Water
Window");
   w.freezeSize (150, 150);

//*****************************************************
//Real-time display window
//*****************************************************
   omRect winRectRt ( w.innerBox.xmin, w.innerBox.ymin + 30,
               w.innerBox.xmax, w.innerBox.ymax);
   RealTimeW w_rt(winRectRt, &w ,"Real-time Display");
   w_rt.bkColor = DARKGRAY;
   w_rt.changeMinimizeIcon( "Real-Time Window" );


//*****************************************************
//the vertical menu under "On Line Acquire":
//*****************************************************
    omVertMenu vmOnLine (2);
    vmOnLine.setMenuFont(menu_font, 1, BLUE);
    vmOnLine + LITERAL_HOTKEYS
         + "~Setup" + vmSetup
         + "S~tart" + LU1( test, &w_rt);

    Adialog = new AnalyseDlg(&w,100,50 );
    Adialog -> usedNew = TRUE;

    omHorizMenu *Principalhm = new omHorizMenu(3);
    Principalhm -> usedNew = TRUE;
    Principalhm -> setMenuFont(menu_font, 1, BLUE);

    omVertMenu Mode(1);
    Mode + LITERAL_HOTKEYS
```

```
      + "~View" + LU1(PassXY2AnalyseDlg, &w) + *Adialog;

    *Principalhm + LITERAL_HOTKEYS
           + "~On-Line-Acquire" + vmOnLine
           + "~Analyse" + LU2(PaintClientArea, &w) + Mode
           + "~Quit" + QuitToDos;

   w + *Principalhm + omTL
       + w_rt;

   *omMEQ + w;

   omMEQ->run();      // run the event queue

   StopGraphics();
   return 0;
}
// END main
```