Upgrading Liquid Metal Cleanliness Analyzer (LiMCA) with Digital Signal Processing (DSP) Technology

by

Xiaodong Shi

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
Montreal, Canada

ISBN 0-612-05476-4

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 deça 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, est une de ces méthodes. Elle permet de mesurer la distribution de taille des inclusions dans les métaux liquides.

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 converti 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. Le système actuel ne peut non plus être utilisé pour des applications qui requièrent un comptage simultané de la distribution de taille des particules et leur distribution dans le temps. Ces limitations retardent la transition du LiMCA à devenir un appareil de contrôle de la qualité des métaux liquides.

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 et de taille des transients classifiés peuvent être obtenu simultanément.

ABSTRACT

The development of advanced metal products requires "clean" liquid metals as their basic materials. There are more and more applications for which the cleanliness of the liquid metals has to be qualified that the number and size of inclusions must be controlled below some acceptable limits. Such demands for quality have resulted in the development of measuring systems that can count the number and size distribution of inclusions. One such device, the so-called LiMCA (Liquid Metal Cleanliness Analyzer), which was developed at McGill University, measures inclusions in liquid metals and has been successfully used in the aluminum industry for years.

LiMCA is based on an 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, which therefore result in transient changes in 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 accumulates the transients in certain time increments.

The current LiMCA system uses analog electronic components to implement the signal processing part. It can only describe a transient by its height or its time of occurrence. This implementation has limited the further development of the system for applications where different types of transients occur and where these transients have to be classified before further processing. The system also limits the applications where the particle size distribution and particle occurrence must be counted concurrently. These limitations have hindered the development of the LiMCA system from an inclusion measuring device into an on-line quality control apparatus.

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 with the help of them, classify it into a certain category. Moreover, it simultaneously counts the classified peaks based on their height and their time of occurrence.

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 study.

Special thanks to Prof. G. Carayannis again for his valuable knowledge of DSP and computer technology that the author learnt from him and applied to the work.

The author would also like to convey his sincere gratitude to Mr. F. Dallaire, the MMPC lab manager, for his willingness to share his valuable experience in LiMCA experimenting and data processing, and for his comments on the thesis.

The author would like to thank Mr. T. Draganovici, a good friend and a valuable colleague of mine, for his daily collaboration and discussions throughout the progress of the work.

Finally, I owe a great deal of debt of gratitude to my wife for her unwavering support and her devotion in raising our lovely daughter.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Preview

The presence of inclusions (i.e. foreign, undesirable particles, such as oxides, intermetallics, etc.) 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 distribution 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 Hutter 81, Mansfield 82] but most of them are off-line techniques that require considerable amount of labour and time. A novel on-line method, known with the acronym LiMCA (Liquid Metal Cleanliness Analyzer) was developed at McGill University by researchers Doutre and Guthrie [Doutre 84]. The principle of operation of the LiMCA system is based on the Electric Sensing Zone (ESZ) Principle (**Section 1.2.1**), which was first developed and applied by Coulter [Coulter 56] to aqueous and organic suspensions at, or near, room temperature.

The LiMCA technique has been successfully used for quality control in the aluminum industry by Alcan, and, being an on-line method, LiMCA has the potential to be used for the development of a process control system. At McGill, a significant amount of research has been carried out for the application of LiMCA to other metals and alloys, such as zinc, magnesium, copper, steel, etc. [Nakajima 86, Kuyucak 89, Kuyucak and Guthrie 89, Lee 91].

In addition to the applications of LiMCA to liquid-metal quality monitoring and control, there were several practices and there are strong desires to use it 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 alloys 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. To know the size distributions and frequencies of occurrence of different types of inclusions at a certain location and a certain time would be of great help to metallurgists studying the processes.

The demand for such a tool for use both in the process study and control motivated our LiMCA research project, which is currently sponsored by FCAR and on NSERC strategic grant. Our final goal is to develop a system that can tell the operator, to some extent, what happened and what is taking place inside liquid metal in various processes. The work described in this thesis involves mainly the work related to the signal processing system of LiMCA. Upon completion, a flexible working platform is provided for further study and development. In the subsequent sections of this chapter, an introduction to the LiMCA system and its operational principle, and the motivations for our work are presented.

1.2. Principle of Operation

1.2.1. Electric Sensing Zone (ESZ) Principle

As mentioned earlier, the theoretical basis of the LiMCA technique is the Electric Sensing Zone Principle (**Figure 1.1**). A conductive liquid medium is separated by an electrically insulated wall. A small opening in the wall submerged in the liquid connects the two parts of the medium. A constant DC voltage is applied across the orifice, while the liquid is forced to flow through it. In **Figure 1.1**, a cross-section view of a cylindrical orifice with length L and diameter D is illustrated. Conductive fluid is flowing through the orifice with constant flow rate Q and electric current I. Because of the geometrical confinement of the orifice, the electric field is intensified inside the orifice and thus becomes very sensitive to the change of the electrical property of the conductive fluid flowing through the orifice. The volume inside the orifice is called the Electric Sensing Zone, **ESZ** for short. When a non-conductive particle passes through the orifice with the fluid flow, the overall resistance of the orifice is increased momentarily and can be detected as a voltage pulse. A non-conductive particle with diameter d suspended in the fluid is shown in **Figure 1.1** as it

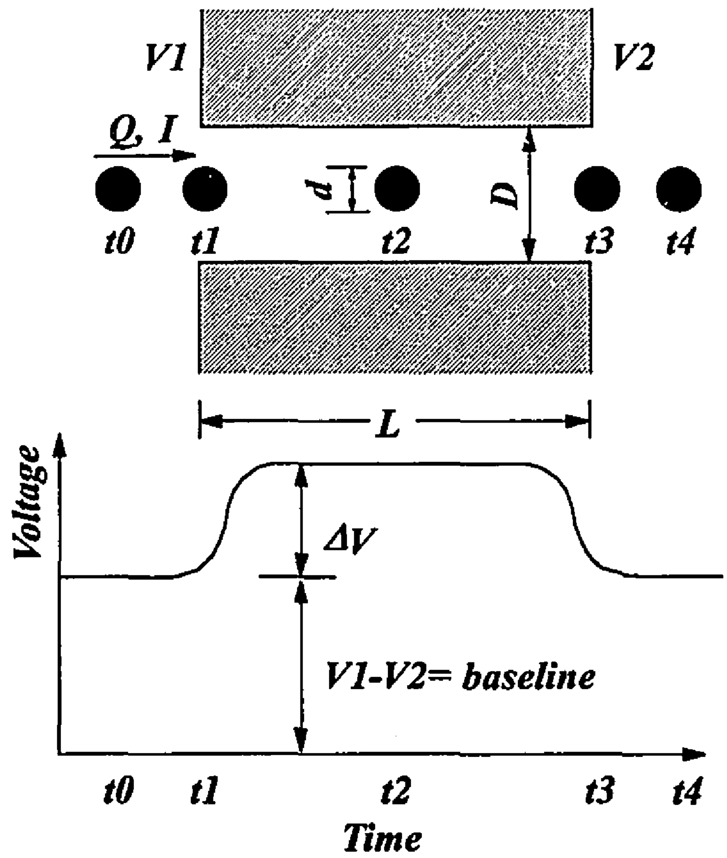


Figure 1.1 Voltage Change due to a Non-conductive Particle

passes through the orifice. The position of the particle is labeled with time ***t1, t2,....*** Under the following assumptions:

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

The voltage change ΔV is related to the volume of the particle by **Equation 1.1** [DeBlois and Bean 70]. This equation is used as a basic relation to predict the size of particle from the voltage change ΔV. A detailed discussion of the ESZ principle can be found in [Doutre 84].

$$\Delta V = I\frac{4\rho d^3}{\pi D^4} f\,(d/D) \qquad (1.1)$$

where

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

1.2.2. LiMCA Sensor and Signal

The LiMCA sensor is designed to have an ESZ of a certain shape and to catch and monitor the voltage change due to a particle passing through the ESZ. The design of the probe and the materials used to construct it depend on the metal or alloy to be evaluated and analyzed.

Figure 1.2 shows a typical LiMCA sensor for use in molten aluminum and its alloys. It consists of an electrically-insulated tube with a small orifice at the side wall near the bottom and two electrodes, one inside, the other outside the tube facing the orifice. The tube is made of Kimax glass, and the electrodes are made of steel. A smoothly-curved orifice is desirable for a stable metal flow through the orifice. This is essential for stable signal. A glass-blowing technique is applied to make the orifice. A

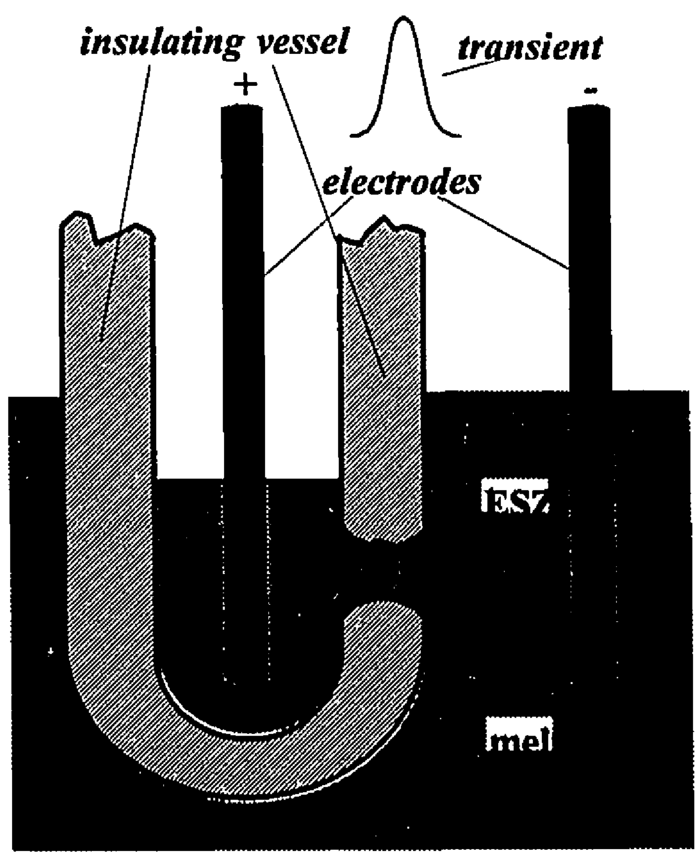


Figure 1.2 LiMCA Sensor

cross-section view of a real LiMCA orifice is shown in **Figure 1.3**. Detailed design parameters can be found both in [Doutre 84] and [Dallaire 90].

In practice, the shape of the orifice clearly violates assumption 3 (cylindrical orifice assumption) that must hold for **Equation 1.1** to be true. Furthermore, in real processes, the shape of particles may not be spherical. Assumption 1 (spherical particle assumption) may also be violated. The work of [Carayannis et al, 92] showed that the cylindrical assumption can be relaxed in that the significant sensitive region of the ESZ of the real orifice is much longer than the size of the inclusions. Although the electric current line distributions throughout the real orifice are quite different from the case of the ideal cylindrical orifice, the streamlines of electric current in the vicinity of the neck of the real orifice are still parallel. Thus it is concluded that the peak values of the resistive pulses (or equivalently voltage pulses) generated from a real orifice and an ideal cylindrical one are equal.

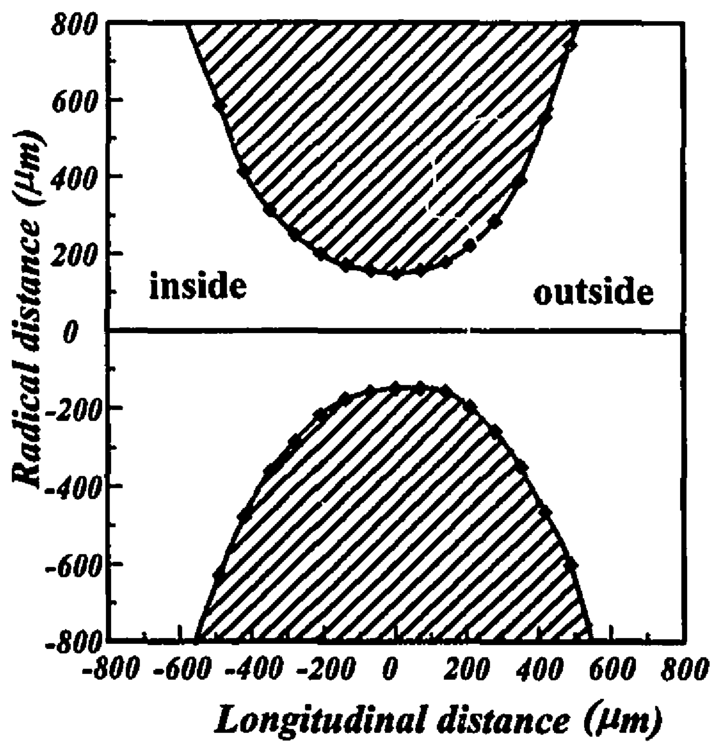


Figure 1.3 A Close-up Longitudinal Cross-section View of a Real LiMCA Orifice

As for the spherical particle assumption, the theoretical modeled resistive pulses of two equal-volume particles of different shapes are shown in **Figure 1.4** [Carayannis et al, 92]. The peaks of the two cases are quite different while the areas below the two curves are equal. From this modeling work, one can conclude that the transient generated by a cylindrical inclusion cannot be easily distinguished from that generated by a smaller spherical inclusion, detected and described only by its magnitude.

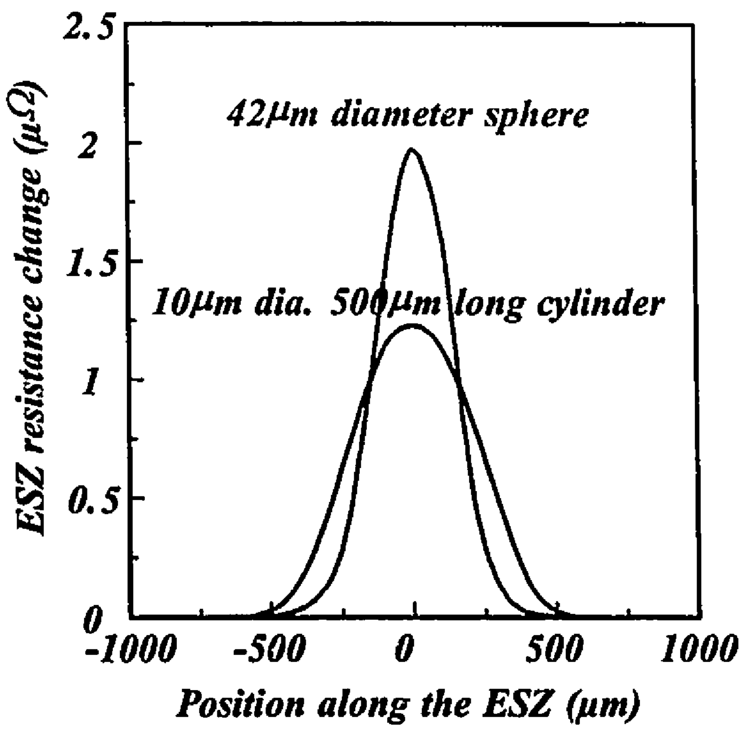


Figure 1.4 Resistive Pulse of Two Equal Volume Inclusions

However, the encouraging fact from this preliminary research is that the shapes of the resistive curves are shown, under certain conditions, to be sensitive to the shapes of the inclusions. The shape information, if extracted, could be used to correct the particle size error due to irregular shape and to identify different types of inclusions. To decode the shape information, further theoretical and experimental studies have to be conducted for a better understanding of the ESZ phenomena. To facilitate the researches, a working platform which can describe the shape of the transient is required. Developing such a platform is the object of this work. In the subsequent sections, the first generation LiMCA system is introduced, its limitations are discussed, and the direction that we take to upgrade it is also presented.

1.2.3. LiMCA System and Signal Processing

The architecture of the first generation LiMCA system, which was designed in the early 80's, is schematically shown in **Figure 1.5**. The system consists of four parts: a sensor (**Section 1.2.2**), a power supply system, a pressure and vacuum 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 signal processing system has two parts, a signal conditioning part and an analog signal processing part. The magnitudes of the voltage transients that the system must detect are in the micro-volt (μV) range and are superimposed on a DC offset which, for a Kimax probe with $300\mu m$ orifice used in molten aluminum, is about 0.1 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 milivolt level for further processing. To increase the sensitivity to small pulses, the signal is also passed through a logarithmic amplifier.

Further processing is carried out by an analog signal processing system, built

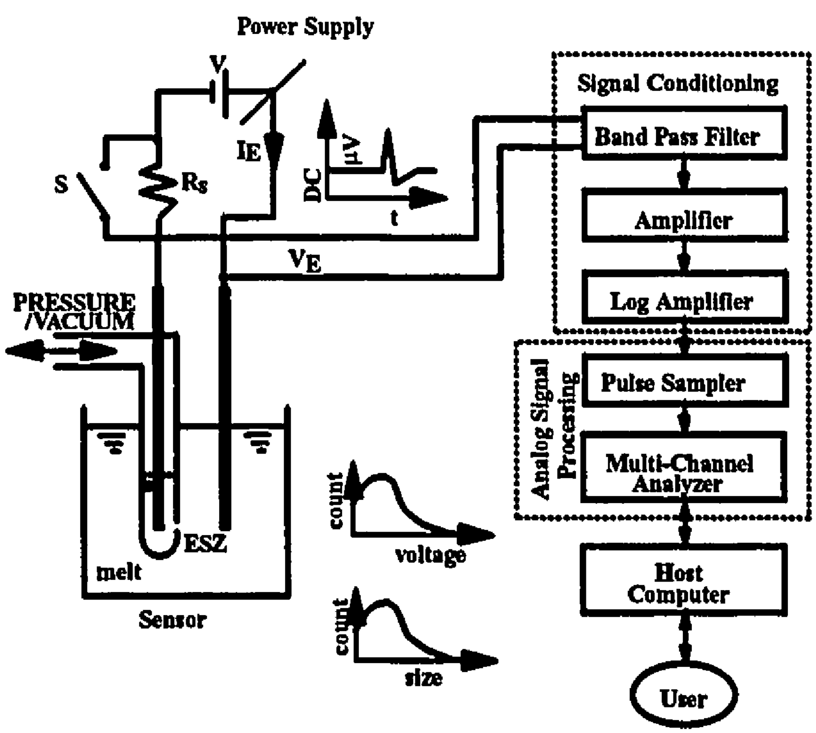


Figure 1.5 Schematic of the First Generation LiMCA System

from commercially available units. Here, a Pulse Sampler (model TN-1246, from Tracor Nortern) is used to detect and measure the height of the transients and feed their magnitudes to a Multi-Channel Analyzer (model TN-7200, also from Tracor Nortern). The latter has two modes of operation, Pulse Height Analysis (PHA) mode and Multi-Channel Scan (MCS) mode, generating a size or a time distribution of the transients (**Figure 1.6**).

In the **PHA** mode, the detected transients are classified according to their magnitudes. Using **Equation 1.1**, this voltage distribution is converted to an inclusion size distribution, which can then be used to calculate measures directly linked to metal cleanliness, such as the number of inclusions per kilogram of metal, the number of inclusions of certain size ranges per kilogram of metal, the volume ratio of inclusions to metal, etc. Among them, one parameter in particular N_{20} is widely used in aluminum industry. It is define as the number of inclusions whose diameter is greater than 20 µm per unit mass of liquid metal. N_{20} is the main output parameter of the industrial LiMCA system. It is obtained assuming that all the detected transients are related to particles and that there is a constant rate of fluid flow through the orifice [Dallaire 90].

In the second mode of operation, the Multi-Channel Analyzer counts the transients that are detected within a certain time increment, treating equally the

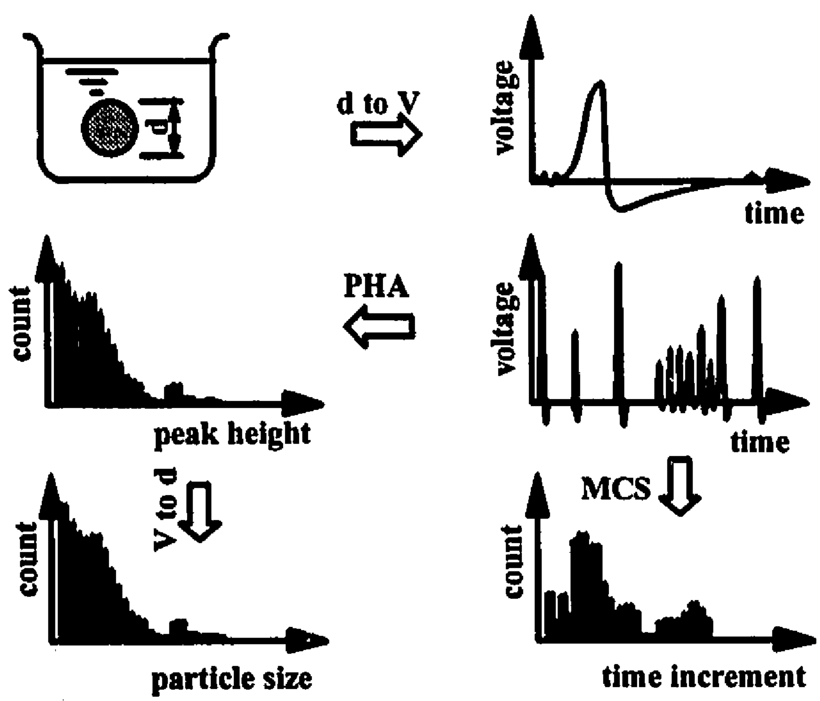


Figure 1.6 LiMCA Data Analysis

transients with different heights. This mode is known as Multi-Channel Scan, or MCS for short. MCS 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 several metallurgical processes, such as, for example, the chlorination and the alloying process of aluminum. These data analysis procedures are illustrated in **Figure 1.6.**

The Multi-Channel Analyzer has an integrated display where these distributions are shown. It is also connected to an IBM-PC through an RS-232 port, and data can be downloaded for future reference and analysis.

1.3. Classes of Real LiMCA Voltage Transients

The reliability of the results from PHA and MCS depends on the accurate peak counts and amplitude measurements of the LiMCA transients. In the LiMCA operations, several types of transients with different characteristics have been observed [Dallaire 90]. They are generated due to different ESZ disturbing factors, and they are not necessarily all caused by inclusions passing through the ESZ. It is obvious that counting and measuring all transients without analysis, introduces errors. Therefore, the types of transients that are caused by inclusions must be first identified and then differentiated from the other types.

In this section, some of the results from our ESZ modeling work will be presented and then the different types of transients that are observed using the LiMCA system will be examined and compared.

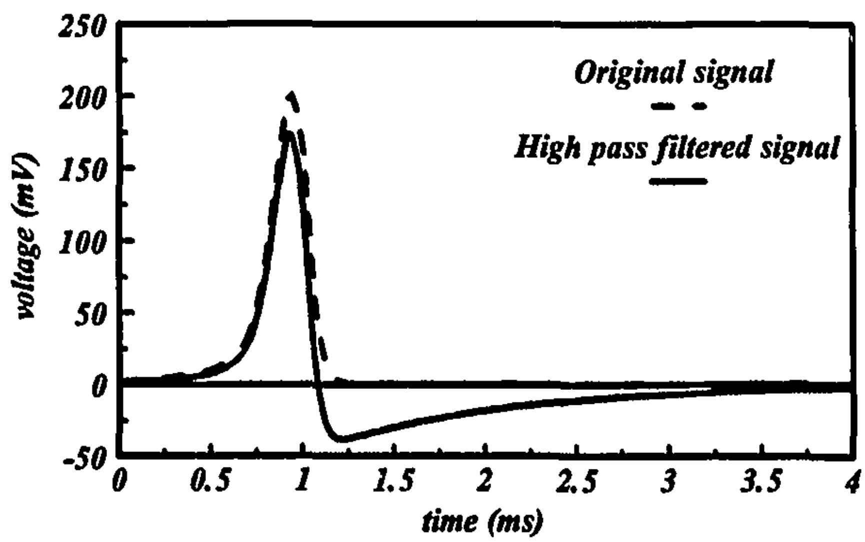


Figure 1.7 Mathematically Modeled LiMCA Transient

1.3.1. Modeling of Real Transient

As mentioned earlier, the first generation LiMCA system uses the relationship developed by [DeBlois and Bean 70] (**Equation 1.1**) to convert the height of the detected transient to the size of the particle that caused it. We also mentioned that this relationship is based on a number of assumptions. In an effort to determine the accuracy of the results generated by the system, We have investigated the sensitivity of the shape and magnitude of the LiMCA transients to these assumptions [Carayannis et al. 92]. In this theoretical study, the behaviour of the ESZ in the presence of a non-conductive particle is mathematically modeled.

Figure 1.7 shows such a modeled transient. The dashed line is the modeled transient generated by the temporary change in the resistance of the ESZ as a spherical, non-conductive particle passes through the orifice. The melt flow is assumed to be laminar and the flow rate constant. This is a reasonable assumption and gives rise to a changing velocity profile across the orifice. Recall that the first signal processing stage is a high pass filter that eliminates the DC component of the signal. The solid line in **Figure 1.7,** shows the effects of this filter with a cutoff frequency of 1 KHz. These include an undershoot following the falling edge of the peak and a magnitude attenuation, which is a function of its frequency components and is usually less then 10% of the magnitude of the peak. The most common observed peaks in typical LiMCA applications are of this type.

1.3.2. Real Transients

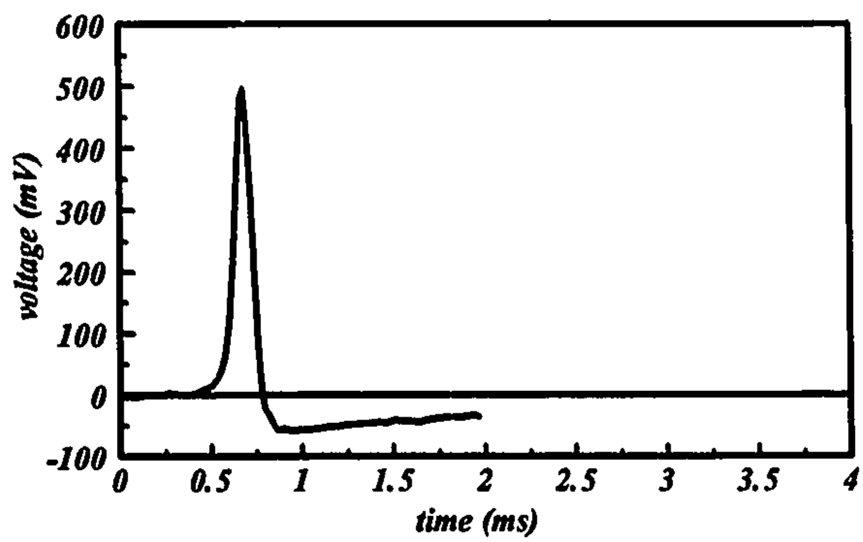


Figure 1.8 A typical Normal Pulse (NP)

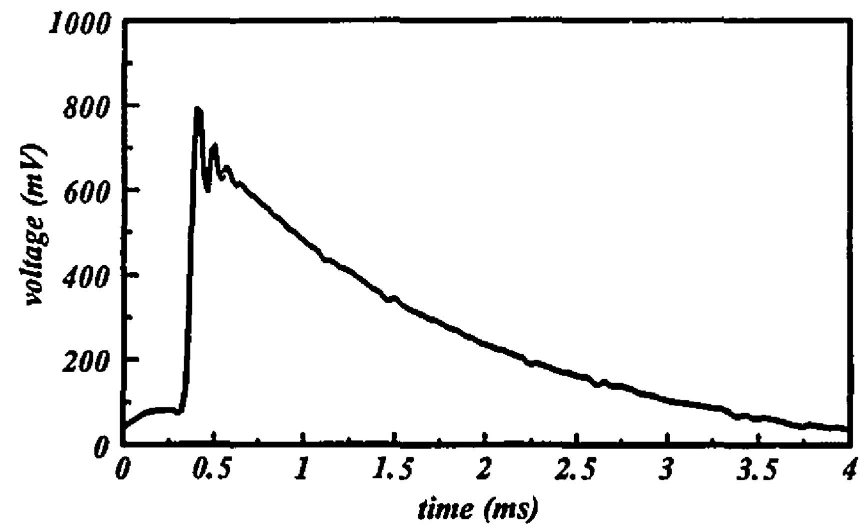


Figure 1.9 A typical Baseline Jump (BJ)

Figure 1.8 shows a transient measured in liquid Aluminum. One can see that the measured transient has similar characteristics with the modeled one, shown in **Figure 1.7**. We call such a signal a Normal Pulse (NP), and argue that it was generated due to the passage of an inclusion through the **ESZ**.

However, other types of transients having different characteristics than normal pulses, have been encountered in aluminum tests, although not as often, under typical operating conditions [Dallaire 90]. Such transients are shown in **Figure 1.9** and **Figure**

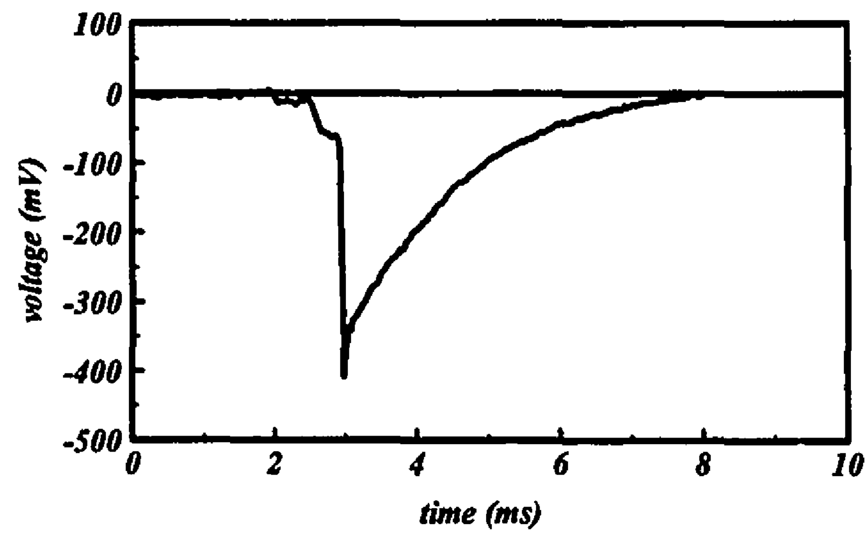


Figure 1.10 A typical Negative Baseline Jump (NBJ)

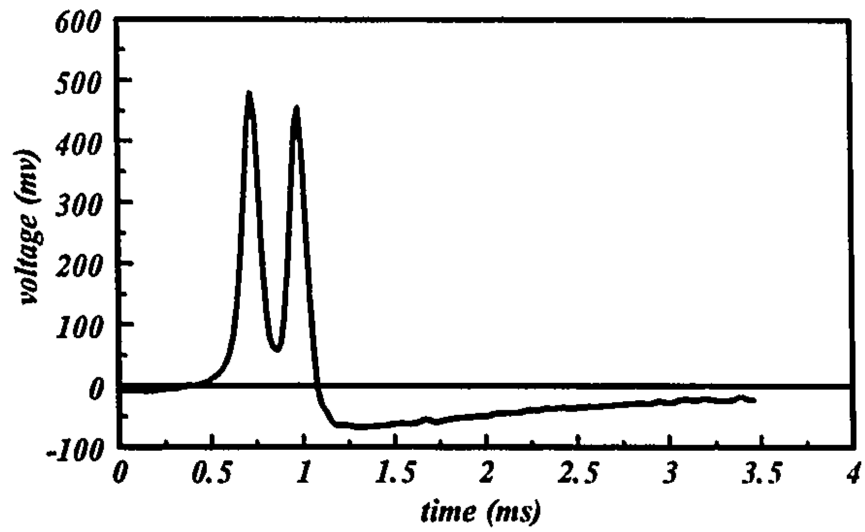


Figure 1.11 A Multiple Pulse (MP)

1.10 and are called Baseline Jump (**BJ**) and Negative Baseline Jump (**NBJ**) respectively. Their characteristics include a steep starting edge and an exponential trailing edge, restoring the baseline. The width (i.e. the time duration) of a **BJ** or a **NBJ** is usually several times larger than that of an **NP** having the same magnitude.

The most prominent physical explanation for the appearance of such peaks, is that they represent the response of the high pass filter 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 blocking or unblocking of the orifice, expansion or shrinkage of the orifice. Furthermore, a long cylindrical inclusion, passing through the orifice in its longitudinal direction, would also give rise to this type of transient.

In rare occasions, when more than one particle pass though the orifice at the same time, transients having more than one peak are detected (**Figure 1.11**). Here two inclusions were present in the **ESZ** at the same time. Such a signal is called Multiple Pulse (**MP**).

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 actual time domain shapes of these two types of signals vary, however their starting slope is quite flat. The presence of such transients indicates oscillations of the baseline (i.e. the magnitude of the DC component) of the signal, and therefore flags improper system operation.

We presented here a summary of the major classes of LiMCA transients. For a comprehensive analysis of the transient classes and related **ESZ** phenomena, see [Dallaire 90].

1.4. Motivations, Methods and Context of This Work

In the first generation LiMCA system, all transients having magnitudes higher than a given noise threshold are detected, their heights are measured and converted to the sizes of the corresponding inclusion particles. However, from our previous discussion it is obvious that only **NP** type transients correspond to particles. **BJ** type transients may be related to particles but in most cases, they are indicative of other **ESZ** phenomena, such as reduced metal flow, partial blockage, orifice size change, etc. It is therefore desirable to develop a LiMCA system that can discriminate and classify the different types of transients. For this purpose, the upgrade of the first generation LiMCA that different types of transients can be differentiated and processed differently became our first objective. The new LiMCA system must also facilitate the research efforts directed to explore the limits of the ESZ technique. It must be designed to provide extensive information, such as the shape and type of the inclusion, the condition of the orifice and the signal etc.

We believe that in order to extract both shape and size information of inclusions from a LiMCA signal, a better understanding of the different **ESZ** phenomena is required. Mathematical modeling, combined with well controlled experiments, can help achieve this. Knowledge of the metallurgical process must be combined with the information obtained from LiMCA in order to identify the possible inclusions (i.e. differentiate expected inclusion particles based on their shape or state, i.e. gaseous, solid, liquid).

The first generation LiMCA system (**Figure 1.5**) 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 objective set above. As a result we considered the design of a software-based LiMCA system using DSP technology.

To ensure compatibility and also facilitate the validation of the new system, our first stage of development is to use DSP technology to develop a new generation, software-based, LiMCA system, functionally equivalent to the first generation one. The second stage is to develop the required code so that the new system can automatically identify the different types of transients. Our 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, categorize each inclusion into one of a number of ***expected classes*** (e.g. based on composition, shape, state, etc.), and to develop a sensor that can be used, not only for quality, but also for process control.

To accomplish our objective, the development of the DSP-based LiMCA can be divided into the following five signal processing tasks. The first task involves sampling the signal and detecting a positive or a negative peak. This is called the ***peak sampling*** process. The second task generates a description of the peak using a number of critical parameters. This is the ***peak description*** process. These parameters are chosen to reflect the characteristics of the different types of transients and the shapes of the inclusions. The ***peak classification*** process is the third task. Here each peak is classified into one of the possible types, on the basis of the parameters used to describe it. In the past, [Thibault et al. 89] investigated the off-line classification of LiMCA signals in the frequency and auto-correlation domains. Although good classification results were achieved, real-time constraints forced us to consider time domain classification algorithms. It was shown that a set of carefully selected measures can enable the design of a fast time-domain classification algorithm [Carayannis and Shi 93].

The forth task extracts the size, shape and volume information of inclusion particles from the peaks classified as NPs in the previous stages. The last task is the development of an intelligent system, which uses the information extracted from the

FROM LiMCA SIGNAL TO PROCESS PARAMETERS

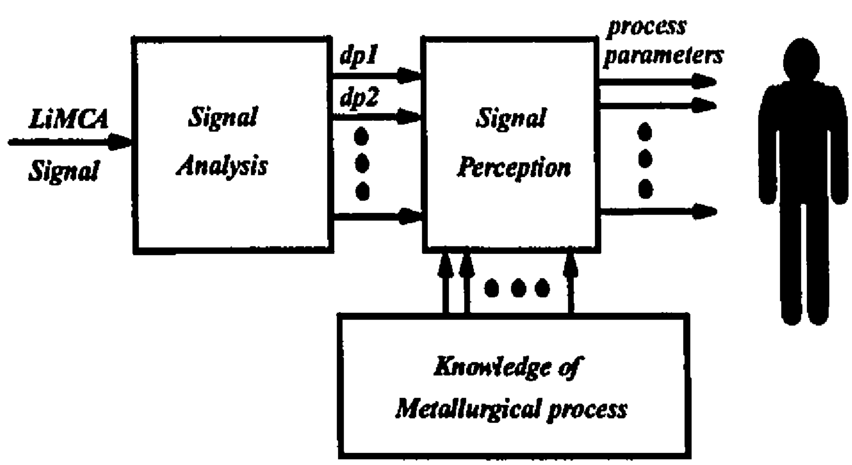


Figure 1.12 Intelligent Signal Analysis

NPs and the frequency of occurrence of other types of transients together with the knowledge about the specific metallurgical processes involved and makes intelligent suggestions to the process operator. **Figure 1.12** schematically shows this process, which is conceptually divided into the ***signal analysis*** stage, that generates descriptions of the detected transients and labels them into associated types, and the ***signal perception*** stage, which identifies the detected particles. The ***signal analysis*** stage involves the first three tasks mentioned above and falls into the scope of this thesis. The ***signal perception*** stage involves the two last tasks and is beyond the scope of this thesis.

In the subsequent chapters, the hardware and the software of the DSP-based LiMCA will be discussed. Finally conclusions of this work and discussions of future developments will be given.

2. DSP-BASED LiMCA SYSTEM

In this chapter, a brief introduction to DSP and a comparison between the digital and the analog signal processing approaches are presented followed by discussions on the particular DSP application for the LiMCA system. An overview of the DSP-based LiMCA is then presented. Finally the hardware environment of the system is presented.

2.1. Digital versus Analog Signal Processing

Before computer technology produced fast and cheap specialized processors, signal processing could only be done by analog circuits. Now more and more applications are implemented digitally. Signal processing generally involves the transformation of signals from one domain to another in real-time (**Figure 2.1**). The purposes of the transformation are to eliminate some unwanted components of the signal (e.g. noise) and to highlight some interesting characteristics that are buried in one domain and can be revealed in other domains.

The differences between analog (**Figure 2.1** (a)) and digital signal processing (**Figure 2.1** (b)) lie in that the former processes signal transformation electronically through an analog electric circuit while the latter carries out the transformation mathematically through a programmable digital circuit (DSP). In the analog signal processing approach, the original signal is processed by dedicated circuits. Then the output of the analog signal process module is either displayed using analog gauges, plotted on paper by an X-Y plotter or more often, nowadays, displayed digitally on a screen and saved on magnetic media. In the case presented in **Figure 2.1** (a) the result of signal processing is digitized and fed into a computer for display and storage. This methodology was used in the design of the first generation LiMCA system shown in **Figure 1.5**, in which commercial analog devices (i.e. Pulse Sampler, Multi-Channel Analyzer) were used.

In the DSP approach in **Figure 2.1** (b) the original signal is first digitized by an analog to digital converter (ADC). Then the signal processing tasks are carried out in a DSP board controlled by software. The software is developed and updated in accord with the signal processing tasks. The DSP board is controlled and monitored by

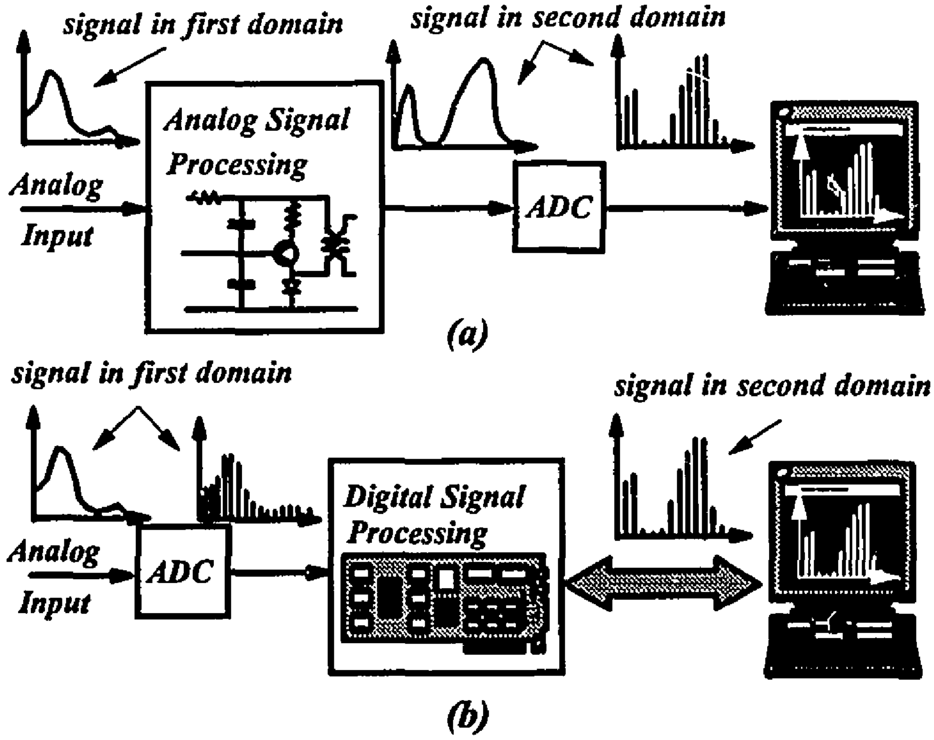


Figure 2.1 Two Signal Processing Approaches: (a) Analog Signal Processing, (b) Digital Signal Processing

a host computer. The results of the digital signal processing are directly uploaded to the host computer through an efficient bi-directional communication channel.

The major advantages of DSP over analog signal processing lie in its flexibility and cost-effectiveness. Digital signal processing is software-based. Thus, it is much easier to be re-configured to accommodate new conditions and parameters. Complicated and newly-developed algorithms can be integrated into the DSP software to improve the overall performance of the signal processing. Such on-going improvements are hard to evaluate and implement with a dedicated analog signal processing circuit. On the contrary, new tasks can be easily added on by modifying the current code and writing more code in the DSP approach. Furthermore, due to its generality, a DSP module is cheaper than an analog signal processing module performing the same tasks.

However, the major concern in the design of a DSP applications is computational power of the selected DSP board, evaluated by computational speed (MIPS, MFLOPS), dynamic range and width of data and address buses. The computational power has always been the limiting factor of the DSP applications with a given hardware. If the speed of calculation is not enough, it will introduce an unacceptable delay for real-time processing. The dynamic range of data buses is critical to the accuracy of the signal processing, and the dynamic range of address buses limits the complexity of the signal processing tasks. In recent years, tremendous efforts have been put into increasing computational power of digital signal processors. As a result, a wide collection of DSP products of different grades are available.

The complexity of the proposed signal processing tasks for the new generation LiMCA clearly suggests that the use of DSP technology is appropriate. The signal processing can be briefly summarize as follows: (see **Section 1.4** for details)

- sample and measure LiMCA peaks by several parameters;
- classify the peaks based on their multi-parameter descriptions.

The real-time peak classification algorithm was not available and is one of the major part of this research. The multi-parameter peak description and the uncertainty of the method used for peak classification contribute to the complexity of the signal processing. Therefore, it is impractical to design and implement an analog signal processing system for the LiMCA signal analysis (including peak sampling, peak description and peak classification) (**Figure 1.12**). Implementing a DSP-based LiMCA system provides a more powerful, flexible and cost-effective solution.

2.2. System Overview

The structure of the DSP-based LiMCA system is illustrated in **Figure 2.2**. Comparing it to the first generation LiMCA system shown in **Figure 1.5**, one can see that the analog components (Log Amplifier, Pulse Sampler, Multi-Channel Analyzer) are replaced by a Digital Signal Processor. This processor is plugged into the bus of a host computer, which is used to interface down to the DSP processor and up to the operator through a newly-developed Graphic User Interface (GUI). The DSP parameters and hardware environment of the system will be discussed in detail in subsequent sections of this chapter. The initialization of the system and the software developed for the system will be discussed in **Chapter 3** and **4.**

2.3. DSP Specifications for LiMCA Application

In order to take full advantage of the DSP technology at minimum cost, the hardware specifications of the selected DSP system should satisfy the requirements of the specific application. Specifically, such specifications as speed, bus dynamic range,

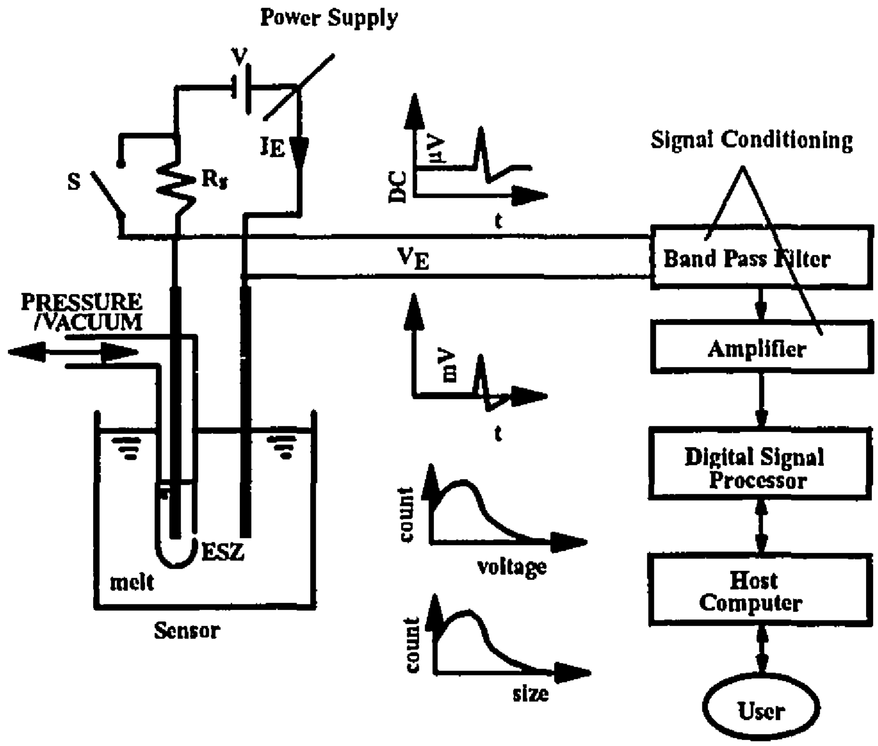


Figure 2.2 DSP-based LiMCA System

ADC sampling frequency and memory size are dependent on the characteristics of the signal to be processed and the required signal processing tasks. Therefore, some preliminary analyses of the LiMCA signal and processing have to be done to set adequate specifications of the DSP processor to be used for LiMCA.

2.3.1. Analyses of LiMCA Signal

Among different types of LiMCA peaks (**Section 1.3**), normal pulses (**NP**) directly relate to inclusions and construct the main stream of the signal. Therefore, the characteristics of normal pulses were taken as the basic feature of the LiMCA signal. The shape of an **NP** is shown in **Figure 1.7** and **Figure 1.8**. In the time domain, the detectable height of an **NP** ranges from 10 µV to 640 µV, in the case of molten aluminum. Considering a noise level of 10 µV under good operating conditions, the Signal-to-Noise ratio is about 36 dB. The duration (width) of an **NP** is around 0.5 ms. Normal pulses have the smallest width among all types of LiMCA peaks. Thus, we define the busiest (worst case) operating condition for the system when the LiMCA signal is purely composed of **NPs** and that they are "chained" together. Under this

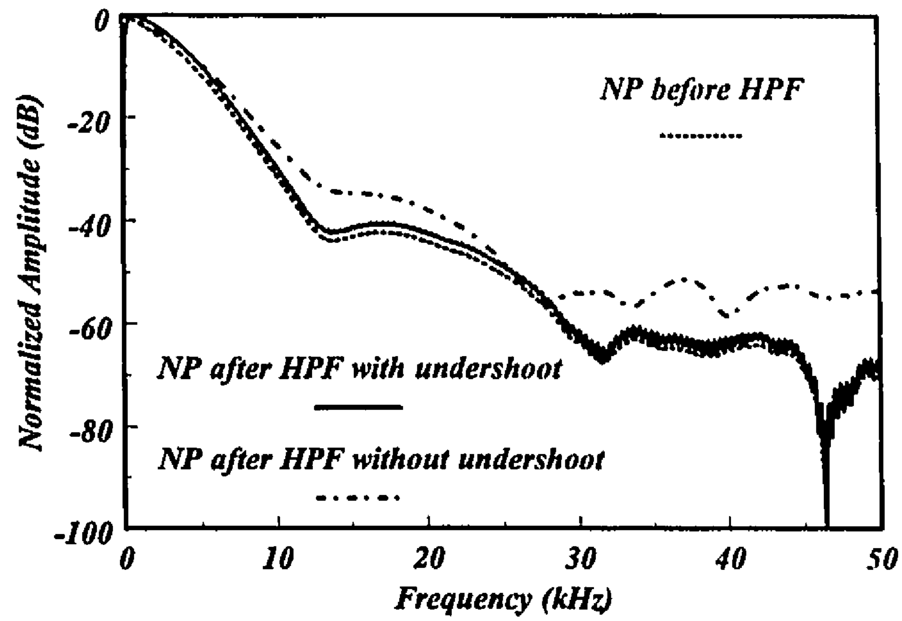


Figure 2.3 Frequency Spectra of the Modeled NP in Figure 1.7

operating condition, the occurrence rate of NPs is approximately 2000 per second. Consequently, in the worst case, 2000 peaks per second need to be processed in real-time. This is clearly an extreme situation which, in real operation, can be observed only for some very short time periods. However this worst case scenario is used as one of the criteria for the design and implementation of the DSP-based LiMCA system.

In the frequency domain, the spectra of the modeled NP (**Figure 1.7**) are illustrated in **Figure 2.3**. The low frequency part of **Figure 2.3** is shown in **Figure 2.4**.

The frequency spectrum labeled with ***NP before HPF*** is the 1024 point radix-2 FFT of the modeled normal pulse taken before the high pass filter. The spectra labeled with ***NP after HPF with undershoot*** and ***NP after HPF without undershoot*** are the FFT of the modeled pulse taken after the high pass filter. However, the way of chopping the pulse in time domain is different in the two cases. The former is chopped at the end of the undershoot when its voltage level restores to zero, while the latter is chopped before its undershoot, when the voltage level reaches zero after its positive peak. The chopping of the positive part of the high pass filtered signal resembles the peak sampling process that we used later to sample both positive and negative peaks. In all three cases, the time domain vectors are expanded to 1024 points by padding the

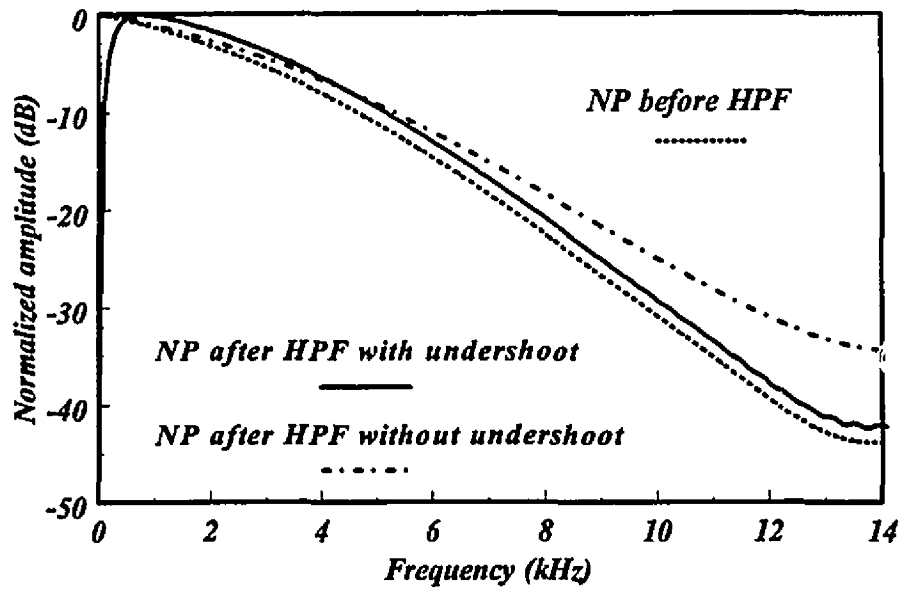


Figure 2.4 **Frequency Spectra of the Modeled NP in Figure 2.3 (low frequency part)**

chopped signals with trailing zeros for FFT. The spectra are normalized before plotting and their amplitudes are measured in decibel (dB).

The frequency spectra of the signal before and after the high pass filter with undershoot are quite close, except for the frequency components below 1 KHz (1 KHz is the cutoff frequency of the high pass filter). While the spectrum of the high pass filtered signal without undershoot is slightly different than those of the other two cases, its shape and tendency are still alike in lower frequency region, up to 25 KHz. The width of the mainlobes of the spectra, in all the three cases, is approximately 14 KHz. Therefore, it is fairly accurate to conclude that the major frequency components of an NP are in the range from 0 to 14 KHz. Other types of LiMCA peaks have narrower frequency spectra than those of NPs [Thibault et al. 89]. Therefore, the bandwidth for NPs automatically satisfies the bandwidth of the other LiMCA signals.

2.3.2. Key DSP Specifications for LiMCA Signal Processing

Based on the analysis of the LiMCA signal presented in the previous section and on the signal processing tasks discussed in **Section 1.2.3** and in **Section 1.4**, some key DSP parameters can be decided.

2.3.2.1. Resolution of Analog-to-Digital Conversion (ADC)

The number of bits used to represent an analog value after the analog-to-digital conversion determines the resolution of the digital representation of the analog signal. Presently, 16-bit analog-to-digital converters are very common and suitable for most applications requiring high precision.

Assuming that the range of the analog signal maps the full range of the ADC input, the absolute quantization error is less or equal to $X_m / 2^B$, where X_m is the full analog input range and B is the number of bits of the analog-to-digital converter [Oppenheim and Schafer, 89]. The relative quantization error is thus within $1/2^B$. For a 16-bit ADC, the maximum relative quantization error is 0.00153%. Neglecting other distortions during ADC, the quantization error gives rise to a Signal-to-Noise ratio of 96 dB, which is much higher than that of the LiMCA signal of 36 dB (Section 2.3.1).

2.3.2.2. Sampling Frequency

The ADC sampling frequency is determined from the bandwidth of the analog signal. In the case of the modeled NP, the frequency range is from 0 to 14 KHz (Figure 2.4). According the Nyquist's Sampling Theorem [Oppenheim and Schafer, 89], the sampling frequency must be equal to or higher than two times the maximum frequency of the analog signal, to avoid aliasing of the high frequencies into the low frequencies, causing distortions. Since the NPs have the widest bandwidth among all

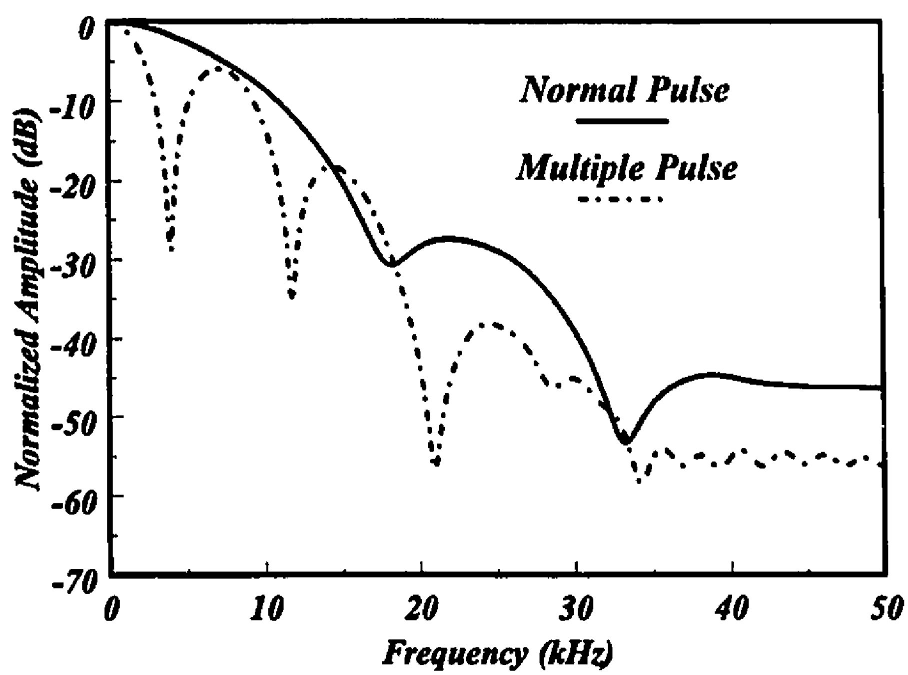


Figure 2.5 Frequency Spectra of a Real NP and an MP

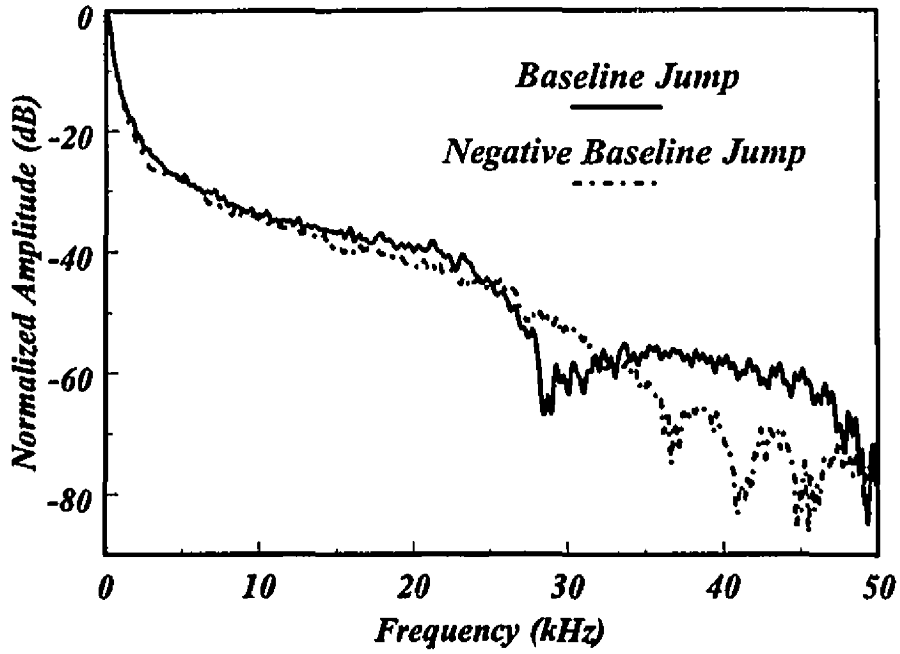


Figure 2.6 Frequency Spectra of a BJ and an NBJ

LiMCA pulses, their maximum frequency is used to calculate the sampling frequency. Considering the Nyquist's Sampling Theorem, the minimum sampling frequency must be equal to or higher than 28 KHz.

Figure 2.5 and **Figure 2.6** show the frequency spectra of real LiMCA transients sampled at 50 KHz. In **Figure 2.5**, the frequency spectra were obtained using a radix-2 FFT of the NP and the MPs shown in **Figure 1.8** and **Figure 1.11** respectively. The spectra in **Figure 2.6** were obtained from the BJ and NBJ signals shown in **Figure 1.8** and **Figure 1.10** respectively. The frequency spectrum of the real NP has the same pattern as those of the mathematical modeled one shown in **Figure 1.7** (time domain) and **Figure 2.3** (frequency domain). The frequency spectrum of the MPs (**Figure 1.11** in the time domain and **Figure 2.5** in the frequency domain) have a "tooth" pattern under the envelope of the frequency curve of the NP. Periodic frequency attenuations occur on the frequency spectrum of the MPs as compared to that of the NP.

The frequency spectra of the BJ and the NBJ are very similar but are evidently different from those of the NP and MPs. Considering the frequency components with normalized amplitudes larger than -30 dB, NPs and MPs have bandwidths about 18 KHz (**Figure 2.5**), which are wider than those of BJs. The minimum sampling frequency to avoid aliasing for normal and multiple pulses must be equal to or higher than 36 KHz (two times their bandwidth according to the Nyquist's Sampling Theorem). To guarantee the accuracy of the signal processing in case the operational conditions change, for example a higher flow rate through the ESZ generates narrower peaks that have wider bandwidth and require higher sampling frequency, some over-sampling is desirable. As a result, the sampling frequency was set to 50 KHz.

2.3.2.3. Input Channels

Some LiMCA applications require real-time measurements at two locations, and the results need to be compared. One example is the evaluation of the filtration of liquid aluminum using a ceramic filter. In this application, two LiMCA sensors are used. One is positioned upstream from the filter and the other downstream from the filter [Tian et al 92]. The results from the two sensors are being compared to calculate the filtration efficiency. To handle this type of applications, the upgraded LiMCA signal processing system must be designed with two parallel processing units, which must operate simultaneously. Therefore, the DSP hardware unit must have two analog input channels, and the processor must be able to process the signals from the two channels in parallel.

2.3.2.4. Computational speed

The speed of a DSP depends upon several characteristics such as clock frequency, instruction set, the length of address and data bus, etc. The required computational speed is considered according to the overall real-time signal processing task and the parameters discussed before. As discussed in **Section 1.4**, the overall task for LiMCA signal processing includes ***peak sampling***, ***peak description*** and ***peak classification*** processes. Moreover, referring to **Figure 2.1**, one can notice that a generic process is always needed for digital signal processing. It is the ***analog-to-digital conversion (ADC)*** process. Considering the sampling frequency of 50 KHz, there are only 20 μs available for all the LiMCA DSP processes between two data samples. Therefore, in order to process LiMCA signals in real-time, the DSP board must be fast enough to guarantee that the processing can be completed within this time constraint.

The clock frequency of the DSP processor can be calculated from the maximum number of clock cycles needed for the execution of the real-time task and the required ADC sampling frequency. However, it is impossible to know the clock cycles needed for the execution of each process before the actual code is written. Nonetheless, a qualitative estimation is still helpful. Assuming that two clock cycles are needed for each instruction, the analysis starts with the number of instructions required. The following discussions are the analysis of each process involved, and the total number of instructions needed for our application are summed up through all the processes.

The ***ADC*** process can be implemented as an interrupt service routine (ISR) triggered by a programmable clock divider at the required sampling frequency. Here, exact timing of the complete DSP application is not necessary. The time difference between the ***ADC*** process and other signal processing processes can be handled using a circular buffer. This design is ideal for complex DSP applications for which manually timing the ***ADC*** process is impossible, as required for some types of DSP systems that ***ADC*** process has to be coded mixed with other processes as a foreground process.

For the interrupt-driven ***ADC*** process, the digitized data are available in a buffer (ADC buffer) when the interrupt occurs. The process moves the data from the ADC buffer to a circular buffer and monitors the buffer status. The instructions that implement the process are:

- 1 *jump to subroutine* instruction to enter the ADC ISR. Program counter (PC) and system status register (SR) are pushed into the system stack;
- 1 *move* instruction to move the digitized data from the ADC buffer to the circular buffer;
- 1 *return* instruction to exit this routine. PC and SR are popped from the system stack;

To manage the circular buffer, the following instructions are needed:

- 1 *move* instruction to fetch the circular buffer write pointer;
- 1 *increment* instruction to increment the pointer one position forward;
- 1 *move* instruction to save the pointer back to memory;
- 1 *move* instruction to fetch the circular buffer read pointer;
- 1 *comparison* instruction to compare the write pointer with the read pointer;
- 1 *conditional jump* instruction following the pointer comparison. The result of the comparison indicates the status of the circular buffer. If it is not overflow, the process returns. Otherwise, it needs extra instructions to flag the buffer overflow error. This is a fatal error that terminates the real-time process. When this occurs, the timing looses its importance. Thus, these extra instructions under this condition are not considered in the real-time timing.

Therefore 6 instructions are needed for managing the circular buffer. For some processors, two registers are needed for *comparison* instructions, and these have to be saved in the system stack before executing the ISR. 4 *push-pop* instructions are needed for this purpose. In total, 13 instructions are required for each ADC channel, about 26 instructions for two ADC channels. Note that for many DSP processors with parallel architecture, parallel data move are allowed. For these, the total number of instructions for two channels can be decreased dramatically. However for a rough estimation, the above analysis is sufficient.

In the ***peak sampling*** process, the same amount of data have to be moved and the circular buffer has to be maintained as for ***ADC*** process. In addition, some *comparison* and *conditional jump* instructions are needed to compare the data fetched from the circular buffer with certain thresholds to find the start or end of a peak. Therefore, for the data movements and buffer maintenance, the same 26 instructions are required. Estimating another 26 instructions for the comparisons, total of 52 instructions are calculated for this process. Therefore, a total of 78 instructions are required for the ***ADC*** and ***peak sampling*** processes together. Note that, these instructions are executed per ADC data sample, i.e., they are executed 50,000 times per second with the sampling frequency set to 50 KHz, resulting in 3,900,000 instructions per second.

For the ***peak description*** and ***peak classification*** processes, due to the complexity of the algorithms used, many more instructions are needed. As a qualitative approximation, the code for the first two processes is estimated as 2% of the total DSP software. This gives rise to about 4000 lines of instructions for the overall DSP task, making it a medium size DSP application. Note that the code written for the ***peak description*** and ***peak classification*** processes is executed per LiMCA peak rather than per ADC data sample as in the cases of ***ADC*** and ***peak sampling*** processes. Considering the worst case operation (2000 peaks per second) (**Section 2.3.1**), 7,844,000 instructions are to be executed in one second for the ***peak description*** and ***peak classification*** processes. In total, 11,744,000 instructions needed to be executed in one second for the overall LiMCA DSP task.

In conclusion, based on the above calculation, the DSP processor must be faster than 12 MIPS (Million Instructions Per Second). Normally two clock cycles are needed for each instruction. Therefore, the clock frequency of the processor to be selected for the LiMCA DSP task must exceed 24 MHz.

2.3.2.5. Summary

In summary, the basic requirement for the DSP board used for LiMCA signal processing includes two input ADC channels with 16-bit resolution, up to 50 KHz ADC sampling frequency, and a DSP with a system clock faster than 24 MHz. As for further enhancement, a DSP processor with parallel architecture is desirable.

2.4. Choice of Hardware Environment

Considering the basic specifications discussed in the previous sections, a DSP-56 co-processor board for IBM PC type computers from Ariel corporation, was selected as the real-time DSP engine. A 50 MHz 80486-based computer is used as the host. The signal processing hardware part of the DSP-LiMCA is schematically shown in **Figure 2.7**. The specifications of the DSP-56 board are summarized in **Appendix A** [Ariel 89].

The DSP-56 is based on the Motorola DSP56001 CPU running at 27 MHz with an instruction cycle time equal to 74.1 nanoseconds. The memory of the processor is arranged in three 64Kx24-bit sections, each with separate address and data

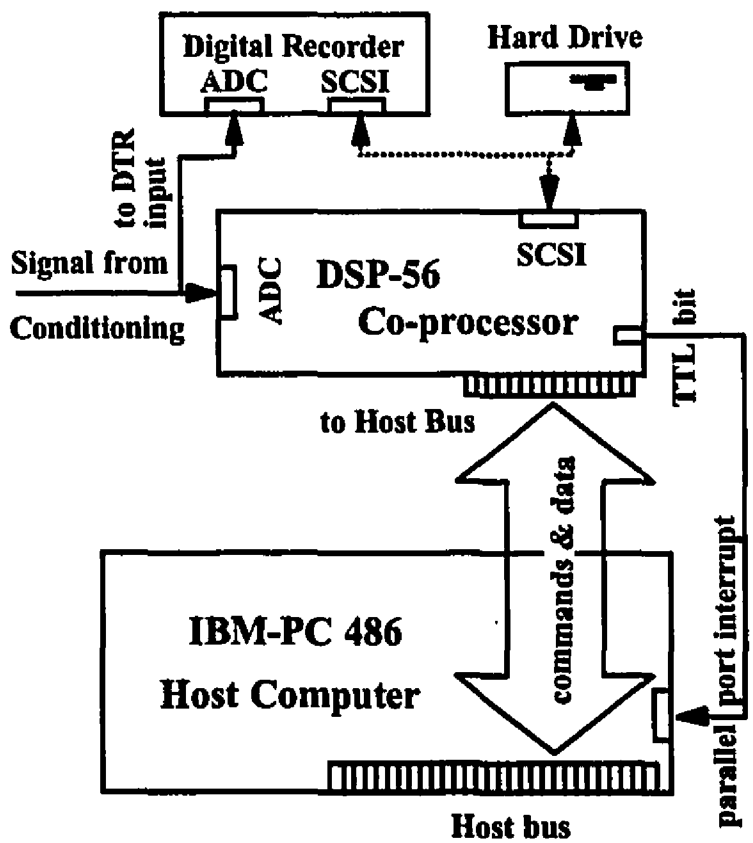


Figure 2.7 Digital Signal Processing Hardware

buses. One section is used for program memory and the other two for data (X and Y data memory). The DSP-56 board has two 16-bit ADC and two 16-bit DAC channels. The sampling rate of the ADC can be selected 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 Standard Interface) interface will be used in the future to save the acquired signal on a hard disk for off-line reference. The DSP-56 also has one input/output bit which we used to interrupt the host computer (using the parallel port interrupt) whenever the real-time DSP process requires attention. The analog signal from the signal conditioning stage is connected to the ADC and to a digital tape recorder (Model RD-101T, from TEAC).

3. SYSTEM CONFIGURATION AND INITIALIZATION

Due to the sophisticated architecture and the required flexibility in the use of the DSP-56 board, its configuration and initialization are not a simple automatic process. Lengthy information and instructions are found scattering in different references [Ariel 89, Motorola 92, Motorola 89]. Therefore, the author found that it is helpful to re-organize and combine information from these references to achieve correct LiMCA operation. For adapting the DSP-56 to other applications, readers must refer to the above mentioned references.

This chapter explains how we customize the DSP configuration settings and parameters that best suit our application. The DSP initialization process and the necessary host function prototypes used to control it are also described.

3.1. The Configuration of the DSP Board for LiMCA

The configuration of the DSP-56 board includes non-programmable

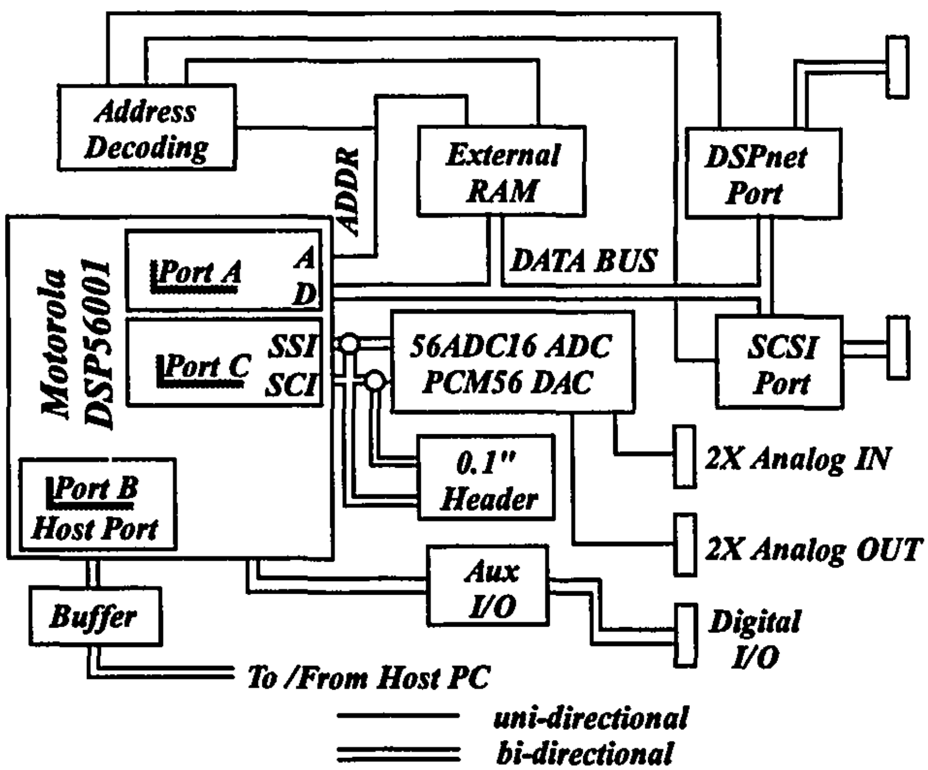


Figure 3.1 DSP-56 Block Diagram [Ariel 89]

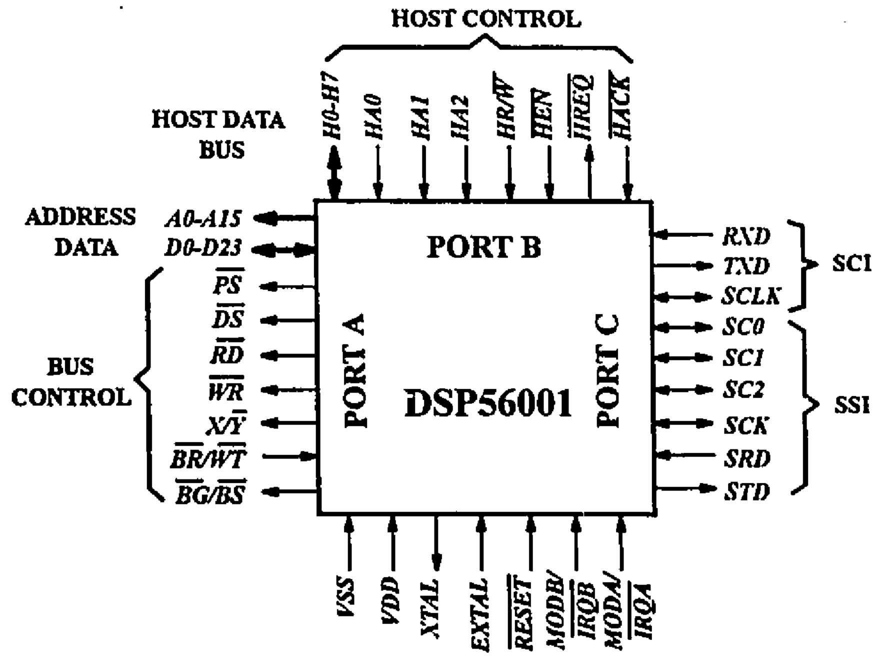


Figure 3.2 Functional Signal Groups of DSP56001 [Motorola 92]

configuration done by setting-up several headers and jumpers on the board, and programmable configuration done by writing appropriate parameters into the dedicated registers. The non-programmable configuration determines the host PC's port addresses for the DSP-56 board, the DSP's memory size, the host PC's DMA (Direct Memory Access) channel, the analog output and the DAC (Digital to Analog Converter) Reconstruction Filter. The programmable configuration sets up the communication parameters of the Port A, Port B and Port C of the Motorola DSP56001 processor, as well as the sampling frequency of the analog interface, the Auxiliary I/O port, the SCSI port and the DSP net port of the DSP-56 board (**Figure 3.1** and **Figure 3.2**).

In the subsequent sections, the non-programmable configuration will be discussed briefly. The programmable configuration of Port A, Port B and Port C, the sampling frequency of the Analog Interface and the Auxiliary I/O port will be discussed in depth. The SCSI port and the DSP net Port will not be considered here, since they are not currently used in our system.

3.1.1. Header and Jumper Settings of the DSP-56 Board

There are several headers and jumpers on the DSP-56 board providing different specifications and usage of the board. Their locations are shown in **Figure 3.3.**

As mentioned earlier, there are five hardware set-ups that need to be configured by setting these headers and jumpers. These are the host PC port addresses, DSP's memory size, the host PC's DMA channel, the analog output and the DAC Reconstruction Filter of the DSP board.

The DSP-56 board is designed as an I/O mapped peripheral that occupies eight I/O port addresses of the host PC, set by means of the jumpers on Header 2. All data transferred to and from the DSP-56 use these I/O port.

The default settings of the jumpers on this header is shown in **Figure 3.4.** The bits are read from the jumper as 1101000 considering the jumpered pairs of pins as 0 and the pairs without a jumper as 1. Adding three trailing zeros to the reading to make it a 10-bit word as 1101000000. This setting corresponds the port addresses $340 through $347. Note that the three trailing zeroes to the reading from header 2 imply that the different selections of the starting addresses of the I/O port are always in multiples of eight. The left most pair of pins is not used and should be left open. At present, the base address of the DSP-56 used for LiMCA is chosen at $340. This is one of the parameters that the host-DSP interface software must know.

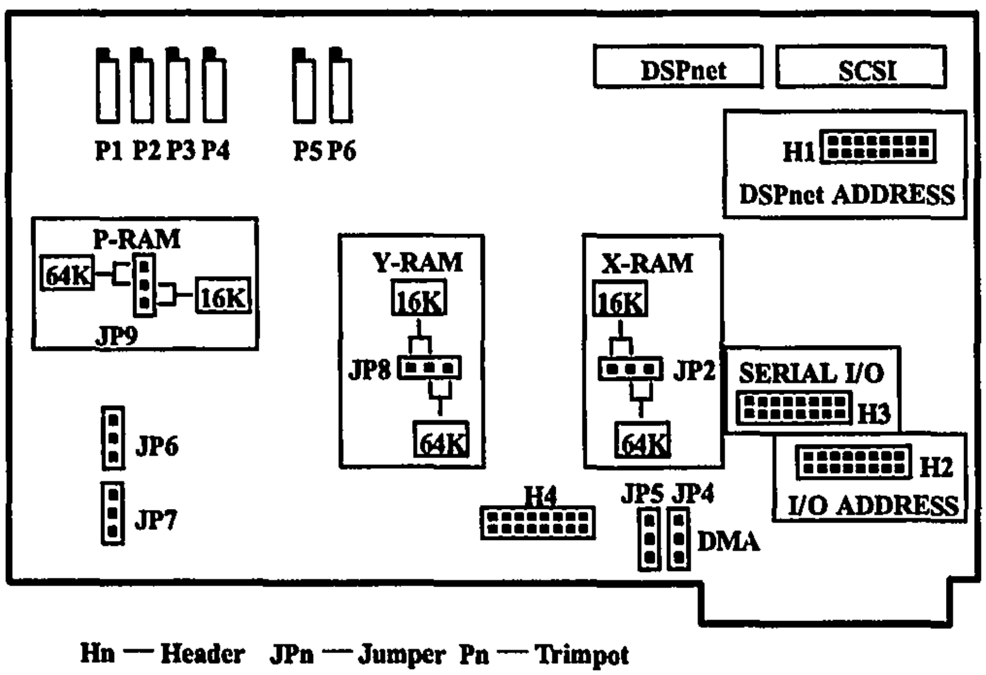


Figure 3.3 DSP-56 Header and Jumper Locations

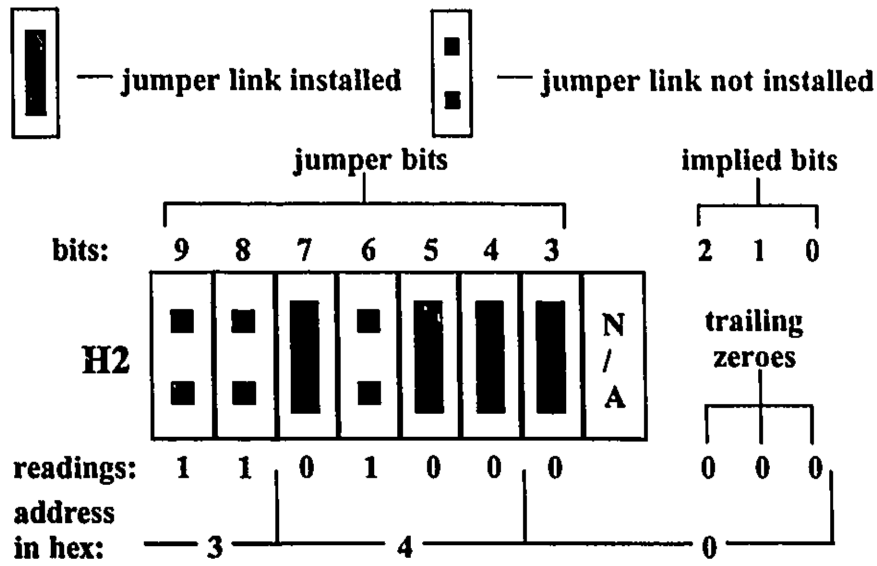


Figure 3.4 Default I/O Address Selection Settings

The memory configuration of the DSP-56 is done by jumpers 2, 8 and 9. The Program memory, and the X and Y data memory can be set for 16K or 64K operation. The jumper locations for the setting of the three memory banks are also shown in **Figure 3.3**. Presently, the size of all three banks are 64K.

Other features of the DSP-56 such as Direct Memory Access (DMA), analog output and the DAC reconstruction filter are not used and are disabled. Besides, the analog input range can be adjusted by two trimpots labeled with "A and B gain" and located near the upper left corner of the board. They provide input gain adjustments over a 17 dB range.

3.1.2. The Configuration of Port A of the DSP56001

As one can see from **Figure 3.1**, the Motorola DSP56001 processor accesses the external memory through its communication port A. This port has 24 data lines, 16 address lines and 7 control lines (**Figure 3.2**). Through this port, the processor can address three blocks of memory, namely program RAM, X and Y data RAM. The size of each memory block, including the processor's internal RAM, can be up to 64K 24-bit words. The external bus timing is controlled by the Bus Control Register (BCR), which is mapped into the X data RAM at X:$FFFE. To synchronize with slower external RAM, zero to 15 wait states can be inserted when the processor accesses the external memory. The number of wait states must be written into the corresponding

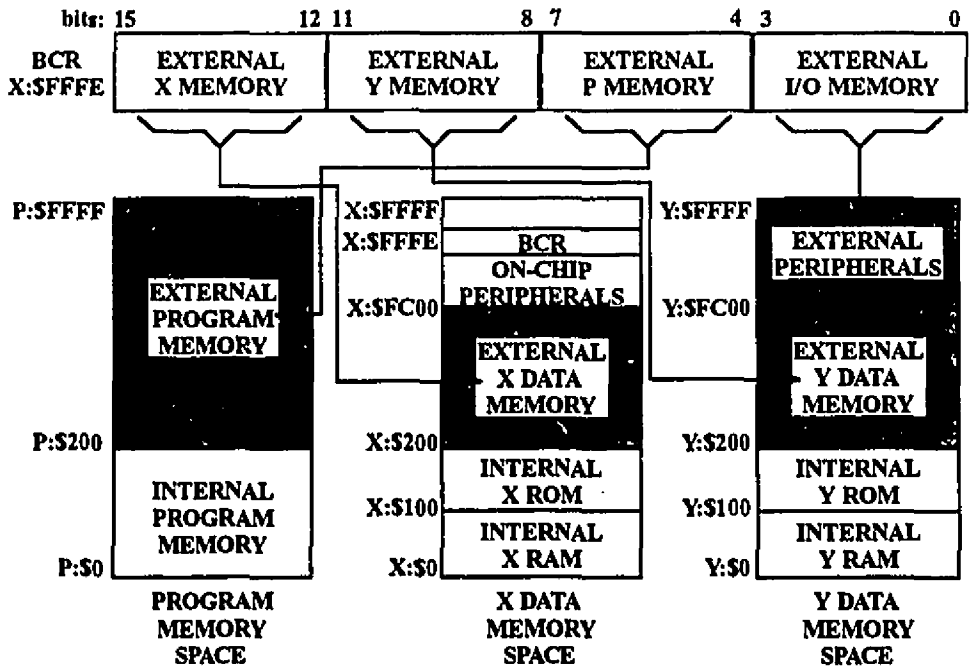


Figure 3.5 Bus Control Register and Memory Spaces

nibble of the BCR (**Figure 3.5**). One wait state is equal to 37 nano seconds for a 27 MHz processor. Note in **Figure 3.5**, that the ROM can be disabled and shadowed by internal RAM.

Following a reset, the DSP56001 processor accesses each of the external memory bank using 15 wait states by default. Since the DSP-56 board uses zero wait static RAM for its external memory. The BCR has to be written with zeroes. The syntax to set zero wait states is "`MOVEP #0, X:$FFFE`".

3.1.3. The Configuration of Port B (Host Interface) of the DSP56001

Port B is a dual-purpose I/O port that can be used as (a) 15 general-purpose pins individually configurable as either input or output pins or as (b) an 8-bit bi-directional host interface (HI) (**Figure 3.2**). For the LiMCA application, this port is configured as a host interface. The selection of HI is done by writing 1 to the Port B Control Register (PBC) at X:$FFE0. This is done by a ROM bootstrap program at booting stage (**Section 3.2.2**).

The HI allows the communication between the host PC and the DSP-56 processor. The communication tasks such as the downloading of DSP programs and

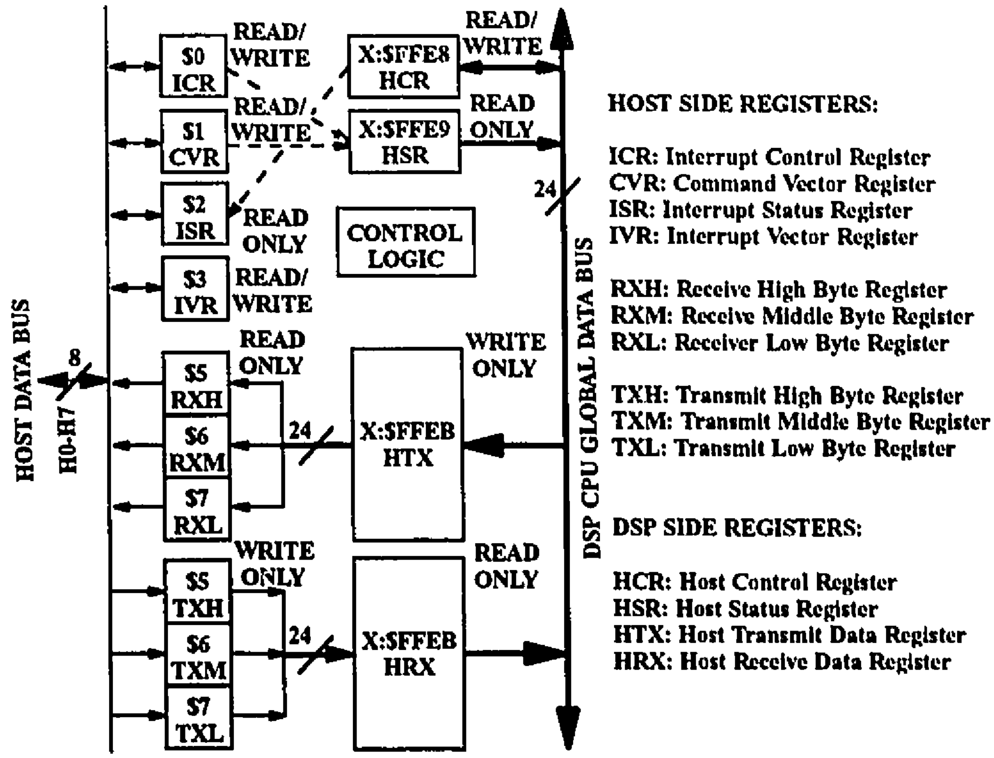


Figure 3.6 Registers of the Host Interface

host control commands from the host PC to the DSP processor and the uploading of real-time data from the DSP board to the host PC, are extensively using the HI during the real-time processing. Therefore, efficient programming of the HI is one of the key factors affecting the overall performance of the DSP-LiMCA System.

The HI is asynchronous and consists of two banks of registers -- one accessible to the host PC and the other accessible to the DSP CPU (**Figure 3.6**, **3.7** and **3.8**). The registers on the host side occupy, in the present configuration, eight 8-bit port locations from $340 through $347 (**Section 3.1.1**) while the registers at the DSP's side are mapped into X memory space occupying 3 memory locations. Note that the port addresses of the registers on the host side, shown in **Figure 3.8**, are the offsets from the base address $347. The HF0 and HF1 bits in the HSR on the DSP side and the ICR on the host side are two general purpose flags for the host to flag the DSP, while the HF2 and HF3 bits in the HCR on the DSP side and the ISR on the host side are similar flags used by the DSP to flag the host PC. The HCP bit in the HSR on the DSP side reflects the status of the HC bit in the CVR on the host side. Data are flowing through the HRX or HTX on the DSP side and the RXH:RXM:RXL or TXH:TXM:TXL triple

	7							0	
X:$FFE8	0	0	0	HF3 (0)	HF2 (0)	HCIE (0)	HTIE (0)	HRIF (0)	HCR (READ/WRITE)

	7							0	
X:$FFE9	DMA (0)	0	0	HF1 (0)	HF0 (0)	HCP (0)	HTDE (1)	HRDF (0)	HSR (READ ONLY)

	23 … 16	15 … 8	7 … 0	
X:$FFEB	RECEIVE HIGH BYTE	RECEIVE MIDDLE BYTE	RECEIVE LOW BYTE	HRX (READ ONLY)
X:$FFEB	TRANSMIT HIGH BYTE	TRANSMIT MIDDLE BYTE	TRANSMIT LOW BYTE	HTX (WRITE ONLY)

BITS IN HCR:

HRIF: Host Receive Interrupt Enable
HTIE: Host Transmit Interrupt Enable
HCIE: Host Command Interrupt Enable
HF2: Host Flag 2
HF3: Host Flag 3

BITS IN HSR:

HRDF: Host Receive Data Full
HTDE: Host Transmit Data Empty
HCP: Host Command Pending
HF1: Host Flag 1
HF2: Host Flag 2

NOTE: The numbers in parenthesis are reset values.

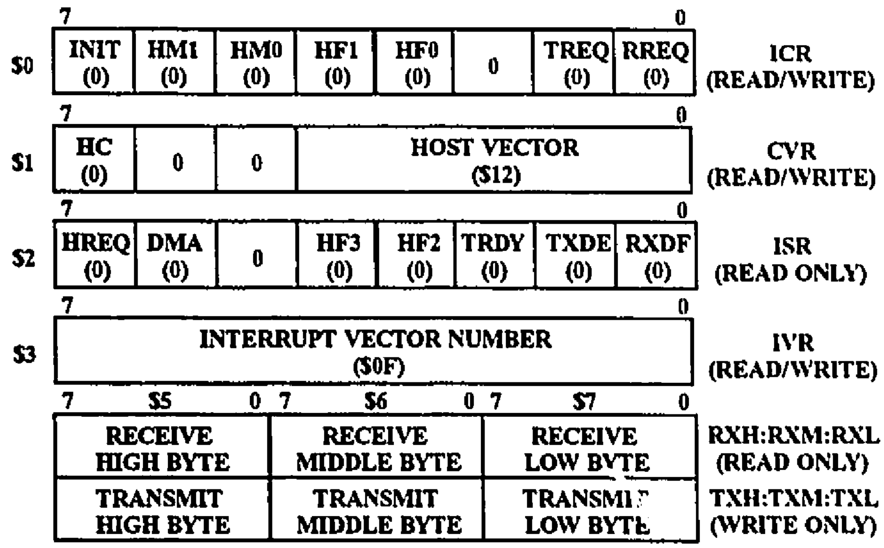
Figure 3.7 HI Registers on the DSP Side

registers on the host side when data transfers are taking place between the host and the DSP-56 board. The HTX and HRX are 24 bit registers located at the same memory location at X:$FFEB and the three register pairs RXH/TXH, RXM/TXM and RXL/TXL are the corresponding three 8-bit registers on the host side. Each pair of the registers share one PC's port address.

The TREQ and RREQ bits in the ISR on the host side are used to determine the DMA mode data transfer direction. The DMA interrupt signal lines DRQ (Data Request) and DACK (Data Acknowledge) are selected via Jumpers 4 and 5 (**Figure 3.3**) [Ariel 89].

Since the DSP-56 board does not have general purpose interrupt sources to the host PC, the interrupt vector number register IVR is never used.

The HI serves as a data transfer passage between the host PC and the DSP and also as a source of interrupt from the host PC to the DSP CPU. It can be programmed to perform data transfer in three modes, namely polling, interrupt and DMA. Only the polling mode of data transfer and the host command interrupt will be discussed in the following sections, since the interrupt and DMA are not used in our present implementation.



BITS IN ICR:

RREQ: Receive Request Enable
TREQ: Transmit Request Enable
HF0, HF1: Host Flag 0 and 1
HM1, HM2: Host Mode Control Bits

BITS IN CVR:

HC: Host Command

BITS IN ISR:

RXDF: Receive Data Register Full
TXDE: Transmit Data Register Empty
TRDY: Transmitter Ready
HF2, HF3: Host Flag 2 and 3
HREQ: Host Request

NOTE: The numbers in parenthesis are reset values

Figure 3.8 HI Registers on the Host Side

3.1.3.1. Data Transfer between the Host and DSP in Polling Mode

In the polling mode, both the host and the DSP processors have to poll certain handshaking flags that regulate the flow of data through the HI. For transfers from the host to the DSP, the host processor polls the TXDE bit in the ISR and the DSP processor polls the HRDF bit in the HSR (**Figure 3.7** and **3.8**). If TXDE is set, indicating that the TXH:TXM:TXL registers are empty, the host processor writes the next data bytes into these data registers. Writing to the TXL results in the TXDE bit in the ISR being cleared. Thus the TXL should always be the last one to write. If TXDE in the ISR is 0, and HRDF in the HSR is 0, data in the TXH:TXM:TXL registers are transferred to the HRX on the DSP side. This data transfer from the host to the DSP sets the HRDF flag in the HSR and thus, it signals that the HRX is full. When the DSP reads the HRX, it clears the HRDF, and this may again initiate a data transfer from the TXH:TXM:TXL triple registers to the HRX (if the TXDE is cleared). In this way, the data transfer continues.

Transferring data from the DSP to the host can be implemented in a similar fashion. Here, the host processor polls the RXDF flag in the ISR and the DSP polls the

HTDE in the HSR (**Figure 3.7** and **3.8**). Writing to the HTX clears the HTDE flag. When the HTDE and the RXDF flags are cleared, data in the HTX is automatically transferred to the RXH:RXM:RXL triple registers and the RXDF flag is set. Reading RXL on the host side clears the RXDF flag. This may again cause another data transfer from the HTX to the RXH:RXM:RXL, and the data flow continues. The following are some sections of programs that implement the host-DSP data transfer.

```
;Host to DSP data transfer by polling at DSP side
DRdy   JCLR   #HRDF, X:HSR, DRdy ;poll HRDF flag in HSR, if not set,
                                 ;data is not ready, poll it again
       MOVEP  X:<<HRX,A          ;if HRDF is set, read HRX

/*Host to DSP data transfer by polling at host side, send a long int to DSP*/
unsigned long data;
register unsigned char *p;
p=(unsigned char *) &data;
while(1)      {
      if(inp(ISR)&TXDE)    //poll TXDE flag in ISR at host side
            break;         //if it is set break the loop and write data to
            }              //TXH:TXM:TXL triple data registers
outp(TXH, *(p+2));         //send the most significant byte first
outp(TXM, *(p+1));         //then the middle byte
outp(TXL, *p);             //the least significant byte should be the last
```

3.1.3.2. Host Command Interrupts

In some cases, the host processor needs to interrupt the DSP process to request immediate service. This can be implemented using the host command interrupt scheme of the DSP56001 processor through the host interface.

As all interrupts of the DSP56001, the host command interrupts are controlled by two registers. One is the Interrupt Priority Register (IPR) at X:$FFFF, the other is the Mode Register (MR) in the Program Controller of the DSP56001.

All the interrupts are associated with an Interrupt Priority Level (IPL). For some of the interrupts, the IPLs are fixed. For the others, the IPLs are programmable and are kept in the IPR. All the interrupt sources and their IPLs are listed in **Table 3.1.** The bit definitions of the IPR and MR are shown in **Figure 3.9**. Two interrupt mask bits in the MR reflect the current processor's IPL and indicate the level needed for an interrupt source to interrupt the processor. Interrupts are inhibited for all IPLs whose value is smaller than the current value of the processor's IPL. Level 3 interrupts always interrupt the processor.

Table 3.1 Interrupt Sources

Interrupt Starting Address	IPL	Interrupt Source
P:$0000	3	Hardware $\overline{\text{RESET}}$ (External)
P:$0002	3	Stack Error
P:$0004	3	Trace
P:$0006	3	SWI (Software Interrupt)
P:$0008	0-2	$\overline{\text{IRQA}}$ (External)
P:$000A	0-2	$\overline{\text{IRQB}}$ (External)
P:$000C	0-2	SSI Receive Data
P:$000E	0-2	SSI Receive Data with Exception Status
P:$0010	0-2	SSI Transmit Data
P:$0012	0-2	SSI Transmit Data with Exception Status
P:$0014	0-2	SCI Receive Data
P:$0016	0-2	SCI Receive Data with Exception Status
P:$0018	0-2	SCI Transmit Data
P:$001A	0-2	SCI Idle Line
P:$001C	0-2	SCI Timer
P:$001E	3	NMI -- Reserved for Hardware Development
P:$0020	0-2	Host Receive Data
P:$0022	0-2	Host Transmit Data
P:$0024	0-2	Host Command (Default)
P:$0026	0-2	Available for Host Command
P:$0028	0-2	Available for Host Command
P:$002A	0-2	Available for Host Command
P:$002C	0-2	Available for Host Command
P:$002E	0-2	Available for Host Command
P:$0030	0-2	Available for Host Command
P:$0032	0-2	Available for Host Command
P:$0034	0-2	Available for Host Command
P:$0036	0-2	Available for Host Command
P:$0038	0-2	Available for Host Command
P:$003A	0-2	Available for Host Command
P:$003C	0-2	Available for Host Command
P:$003E	0-2	Illegal Instruction

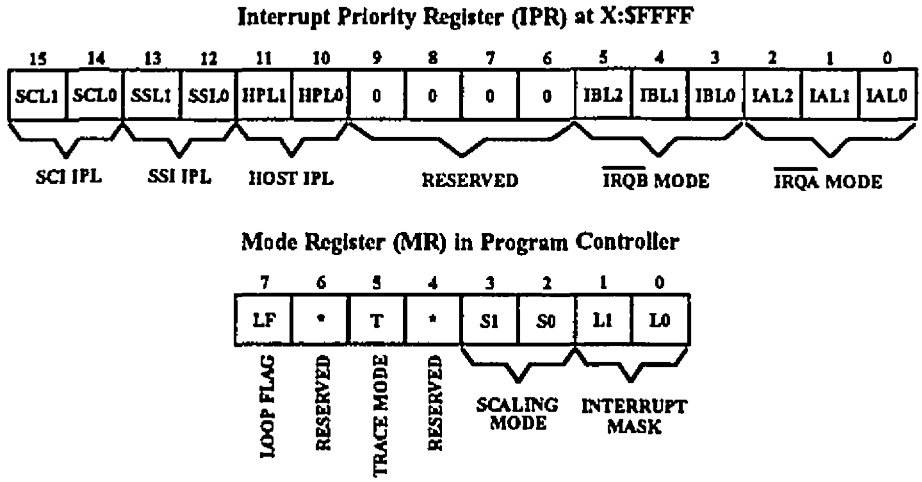


Figure 3.9 Interrupt Priority Register and Mode Register

From **Table 3.1**, one can see that each interrupt source is vectored (one of 32 vectors) to a separate, fixed, two-word service routine located in the lowest 64 words of the program memory. The host interrupt vectors are from P:$0024 through P:$003C.

The programming procedures of the host command interrupt and sample programs can be summarized as follows:

- Shut off all interrupts but level 3 interrupts by setting the L0 and L1 bits in the MR (**Figure 3.9**);

```
ORI     #$11,MR
```

- Set the IPL for the HI by choosing a combination of the HPL0 and HPL1 bits in the IPR (**Figure 3.9**);

```
BSET    #HPLO, X:<<IPR
BSET    #HPLO, X:<<IPR          ;set host IPL to 2
```

- Set up the pointer for the corresponding interrupt service routine. This is done by writing '`JSR START_HOST_ISR`' followed by a '`NOP`' command into the two-word interrupt vector spaces. `START_HOST_ISR` is the starting address of the interrupt service routine residing in the low program memory for the fastest servicing.

```
ORG     P:$0024                 ;default host interrupt vector
JSR     START_HOST_ISR          ;jump to the interrupt service routine
NOP                             ;use a do-nothing operation
                                ;to eliminate pipeline effect
```

- Set the HCIE bit in the HCR (**Figure 3.7**) to enable host command interrupt.

```
BSET    #HCIE, X:<<HCR
```

- Start host interrupts by manipulating L0 and L1 bits in the MR to lower the processor's IPL (**Figure 3.9**).

```
ANDI   $FC, MR        ;clear L0 and L1 bits in MR to enable
                      ;interrupts
```

The host can then write the host vector in the CVR of the HI and set the HC bit of the register (**Figure 3.8**). Note that the actual value of the host vector should be one half of the corresponding interrupt vector in **Table 3.1**. For example, the host vector should be $12 for host command interrupt $24. Setting the HC flag in the CVR causes the HCP bit in the HSR to be set and starts the Interrupt Service Routine from the location in the Interrupt Vector Table, corresponding to the host vector in the CVR.

3.1.4. The Configuration of Port C of the DSP56001

The Port C interface of the DSP56001 is a triple-function I/O port with nine pins (**Figure 3.2**). Three of the nine pins can be configured as general-purpose I/O or as the serial communications interface (SCI) pins, and the other six pins can be configured as general-purpose I/O or as synchronous serial interface (SSI) pins. However, in the implementation of the DSP-56 co-processor board, this interface is used as the SSI to interface to the ADC and DAC circuitry (**Figure 3.1**). Therefore, this port should only be configured as the SSI.

The SSI of the DSP56001 has three dedicated I/O pins (**Figure 3.2**), which are used for transmit data (STD), receive data (SRD) and serial clock (SCK). Three other pins may also be used, depending on the mode selected; they are serial control pins SC0, SC1 and SC2.

The configuration of Port C is controlled by the Port C Control Register (PCC) at X:$FFE1 (**Figure 3.10**). Writing $1F8 to the PCC configures Port C as an SSI and the remaining 3 pins as general purpose I/O, as required by the DSP-56 co-processor board.

The SSI can be viewed as two control registers (CRA and CRB), one status

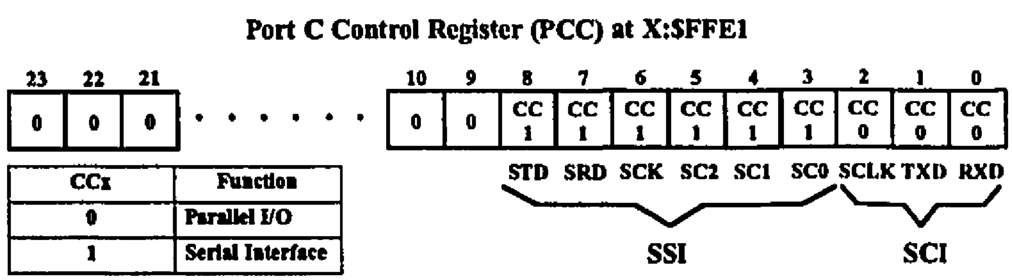


Figure 3.10 Port C Control Register (PCC) and Configuration

register (SSISR), a transmit register (TX), a receive register (RX) and a special-purpose time slot register (TSR). Among them, the RX and TX share one memory location at X:$FFEF, while the SSISR and TSR share another location at X:$FFEE (**Figure 3.11**).

The CRA and CRB control the SSI. The flags in the SSISR can be used for polling purposes. The RX and TX are 24-bit data registers for data transfer from the ADC to the RX or from the TX to the DAC. The most significant 16 bits of the two registers are used for 16-bit ADC and DAC. The least significant 8 bits of the two registers are not used and are automatically filled with zeroes during the data transmission.

Since a dedicated ADC and DAC circuit is connected to the SSI, some of the bits in the CRA and CRB are fixed in accordance with the requirement of the circuit. These bits should be initialized accordingly and not be modified in any circumstances.

In the CRA, bits DC4 to DC0 must be set to 2, i.e. 00010 in binary, for two words per clock frame in network mode. This setting is essential for two ADC and/or DAC channels working simultaneously (see the timing diagram in **Figure 3.12**). Bits

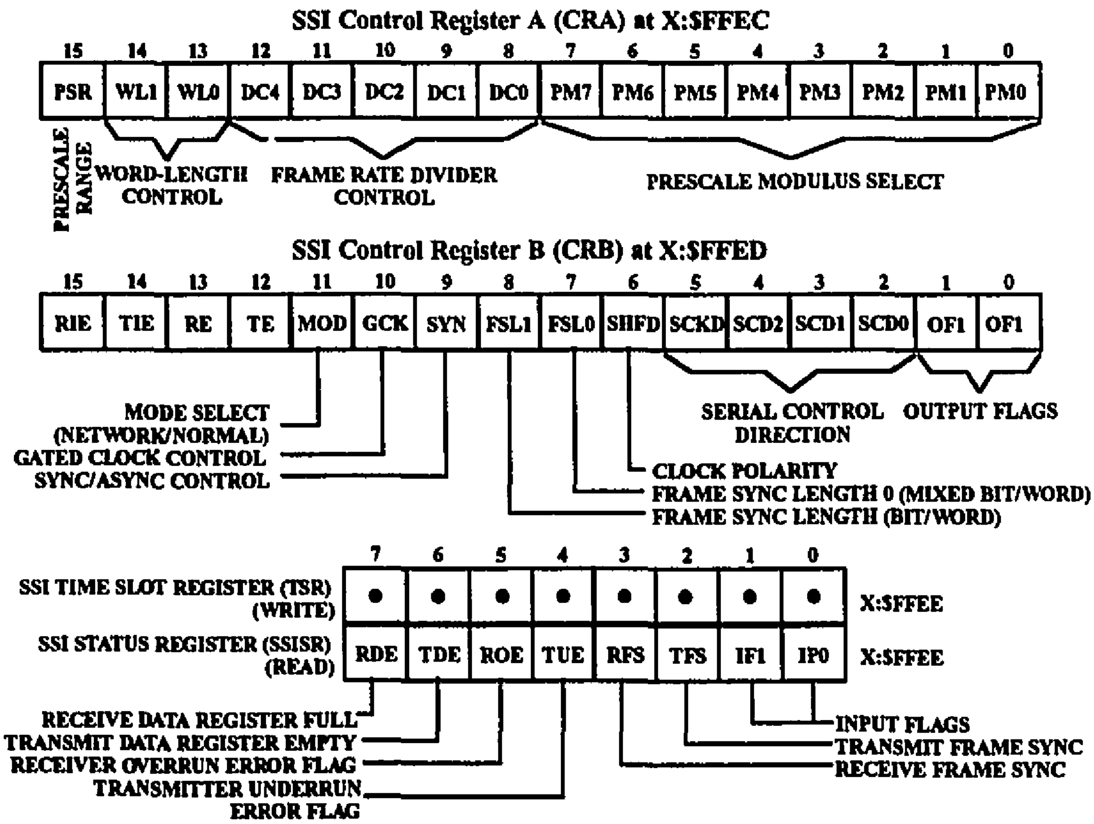


Figure 3.11 SSI Control and Status Registers

WL1 and WL0 must be also set to 10 (in binary), to select a 16-bit word length for the 16-bit ADC and DAC. The other bits in CRA should be set to zero. In summary, 0100,0010,0000,0000 in binary format or $4100 in hexadecimal format should be written into the CRA for the simultaneous use of the ADC and DAC sections.

In the CRB, bits OF1 and OF0 are output flags. At the initialization stage, they have no effects. The serial control direction bits, SCD0, SCD1, SCD2 and SCKD are fixed for the ADC and DAC circuitry, with SCD0 equal to 1 and the rest equal to 0, to configure SC0 as an output pin. Bits FSL1 and FSL0 must be cleared to select a word-length frame clock synchronization for the word length specified by the WL1 and WL0 bits in the CRA. The SYN bit should be set, to select synchronous mode, and the GCK bit cleared, to select a continuous clock. The MOD bit must be set, to configure the SSI in network mode. This mode enables the DSP56001 to receive two 16-bit word frames from the ADCs and send the same number frames to the DACs (see the timing diagram in **Figure 3.12**). Therefore, both channels of the ADC and DAC can be activated at the same time. As a result, the lower 12 bits of the CRB should be configured as 1010,0000,0100 in binary or $A04 in hexadecimal. Bit 12 to bit 15 of the CRB are enable bits. The TE bit enables the transfer of data from the TX to the transmit shift register and the RE bit enables the transfer of data from the receive shift register to the RX. The TIE bit enables the transmit interrupt at P:$0010 (SSI Transmit Data) and P:$0012 (SSI Transmit Data with Exception Status) on the condition that the TX is empty and the transmit shift register is not empty for the P:$0012 interrupt, or on the condition that the TX is empty and the transmit shift register is empty for the P:$0010 interrupt (**Table 3.1**). The RIE bit enables the receive interrupt at P:$000C (SSI Receive Data) and P:$000E (SSI Receive Data with Exception Status) on the condition that the RX is full and the receive shift register is empty for the P:$000C interrupt, or on the condition that the RX is full and the receive shift register is also full, for the P:$000C interrupt. These bits can be toggled to enable or disable the associated interrupts. However, the TE and TIE, and the RE and RIE should be set or cleared in pairs. If both the DAC and ADC channels are used, all these bits have to be set to 1 to enable all the SSI interrupts. In summary, $FA04 should be written into the CRB when all the DAC and ADC channels are being used.

The data transfer from or to the SSI are carried out by interrupt service routines. The interrupt vectors for the SSI start from P:$000C to P:$0012 (**Table 3.1**). One sample of the SSI interrupt service routine from Ariel, shown below, demonstrates a simple way to service the SSI data transfer (The timing diagram and data flow of this sample program are illustrated in **Figure 3.12**):

```
datain      jclr        #3, X:<<M_SR, chan_B
            movep       X:<<M_RX, a1
            bclr        #0, X:<<M_CRB
            movep       a1, X:<<M_TX
            rti
Chan_B      movep       X:<<M_RX, b1
            bset        #0, X:<<M_CRB
            movep       b1, X:<<M_TX
            rti
```

where M_SR stands for the address of the SSISR, M_RX for the address of the RX, M_CRB for the address of the CRB and M_TX for the address of the TX.

This routine services both the ADCs and the DACs using only the SSI Receive Data Interrupt and the SSI Receive Data Interrupt with Exception Status. The entry point of the routine P: <datain should be installed at both P:$000C for SSI Receive Data and P:$000E for SSI Receive Data with Exception Status, if there is no separate error handling interrupt service routine used for the P:$000E interrupt.

The active ADC channel is determined by polling bit 3, the RFS bit, of the

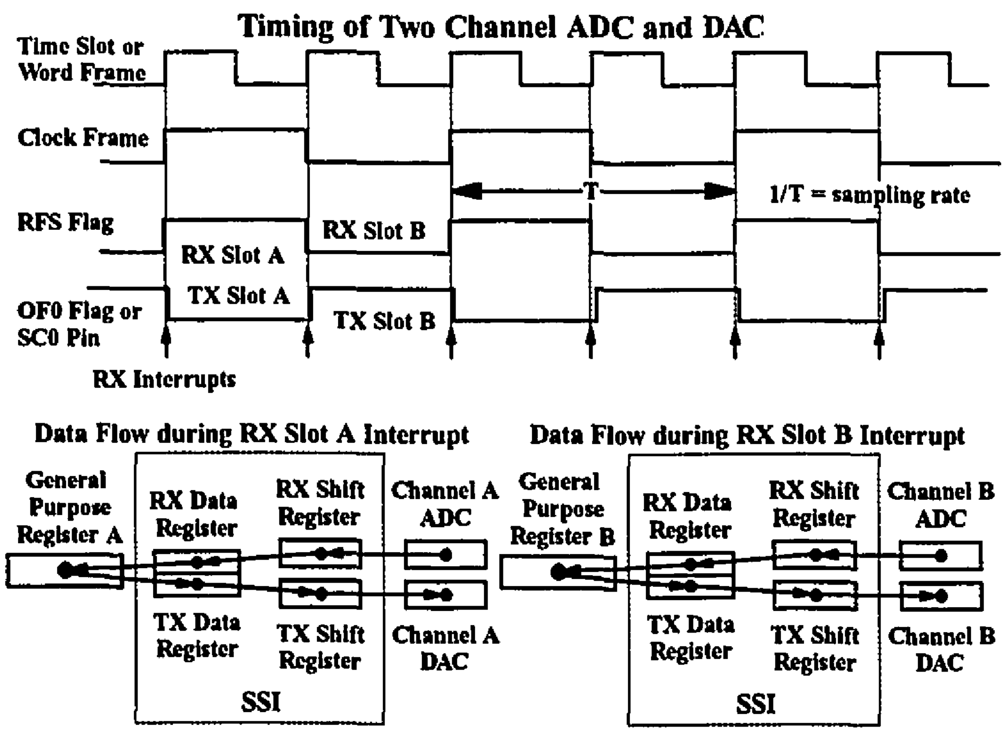


Figure 3.12 Timing Diagram and Data Flow of the Simultaneous Uses of ADC and DAC of Both Channels Using SSI Receive Data Interrupts

SSISR. As we discussed earlier, the SSI is configured in synchronous network mode and two time slots per one clock frame (DC4, DC3, DC2, DC1 and DC0 in the CRA are 00010). This configuration indicates that at each time slot, a word is transmitted into the RX. Thus, two words are received in one clock frame. The data from channel A of the ADC are gated into the RX at the time slots when the clock frames occur and the data for channel B of the ADC are gated into the RX at the time slots when the clock frames do not occur. The status of the clock frame is reflected by the RFS bit in the SSISR. Therefore, this flag is polled to determine the active ADC channel in the above sample program.

The SC0 pin is used to select the DAC channel. SC0 low selected DAC channel A and SC0 high selects DAC channel B. The status of Bit 0 or the OF0 bit of the CRB controls the status of the SC0 pin. Therefore, this bit is used to toggle between the two DAC channels. Writing a word to the TX services the DAC on channel A when OF0 bit is cleared, or channel B when OF0 bit is set.

In summary, similar procedures, as with the HI (**Section 3.1.3.2**), should be undertaken for the use of SSI. They are listed below with sample assembly instructions:

- Shut off all interrupts but level 3 interrupts by setting the L0 and L1 bits in the MR (**Figure 3.9**);
```
ORI    #$11, MR
```
- Write a number to the TX register to turn on the SSI:
```
CLR    A
MOVEP  A, X:<<TX
```
- Initialize the SSI as needed by writing to the CRA and the CRB accordingly;
```
MOVEP  #$4100, X:<<CRA
MOVEP  #$FA04, X:<<CRB
```
- Set up Port C Control Register (PCC) to enable the SSI;
```
MOVEP  #$1F8, X:<<PCC
```
- Set the IPL for the SSI by choosing a combination of the SSL0 and SSL1 bits in the IPR (**Figure 3.9**);
```
BCLR   #SSL0, X:<<IPR
BSET   #SSL1, X:<<IPR       ;set SSI IPL to 1
```
- Set up the pointer for the corresponding interrupt service routine. This is done by writing '`JSR START_SSI_ISR`' followed by a '`NOP`' command into the two-word interrupt vector spaces for the SSI interrupts. `START_SSI_ISR` is the starting address of the interrupt service routine residing in the low program memory for the fastest servicing.
```
ORG    P:$000C              ;SSI Receive Data interrupt vector
JSR    START_SSI_ISR        ;jump to the interrupt service routine
NOP                         ;use a do-nothing operation
```

```
                                    ;to eliminate pipeline effect
        ORG     P:$000E             ;SSI Receive Data interrupt with
                                    ;Exception Status
        JSR     START_SSI_ISR       ;jump to the interrupt service routine
        NOP                         ;use a do-nothing operation
                                    ;to eliminate pipeline effect
```

- Set up sampling frequency for the ADC (see next section);

```
        MOVEP   #$900000, Y:<<MCR
```

- Start the SSI interrupts by manipulating L0 and L1 bits in the MR to lower the processor's IPL (**Figure 3.9**).

```
        ANDI    $FC, MR             ;clear L0 and L1 bits in MR to
                                    ;enable interrupts
```

3.1.5. Selecting Sampling Frequency of the Analog Interface and Using the DSP Auxiliary I/O Port

The sampling frequency of the ADC and the use of the Auxiliary port are controlled by the Mode Control Register (MCR) at Y:$FFF0. The frequency is selected through bits 23 to 20 of the MCR. The combinations of these bits and the associated sampling frequencies are listed in **Table 3.2**. Bit 19 of the same register controls the auxiliary I/O output line and bit 18 toggles the ADC mode between the Normal 16-bit mode and the High Speed 12-bit mode. The remaining bits of the register should always be written with zeroes.

Table 3.2 Sampling Frequency Selections [Ariel, 89]

Bits in MCR 23	22	21	20	Sample Rate (KHz) in Normal Mode	Sample Rate (KHz) in High Speed Mode
0	0	0	0	32	128
0	0	0	1	16	64
0	0	1	0	8	32
0	0	1	1	4	16
0	1	0	0	2	8
1	0	0	0	100	400
1	0	0	1	50	200
1	0	1	0	25	100
1	0	1	1	12.5	50
1	1	0	0	6.25	25
0	1	0	1	22.05	88.2
0	1	1	0	44.1	176.4

Writing 1 to bit 19 outputs a TTL high level to the auxiliary port and writing 0 outputs a low TTL level. Bit manipulation commands should not be used here. The execution of such command at bit 19 unpredictably disturbs the sampling frequency. Therefore, the MOVEP command should always be used to update the content of the MCR. For example, to set the sampling frequency to 50 KHz and output a TTL high at the auxiliary I/O port, one should write:

```
MOVEP #$980000,Y:<<MCR
```

to output a TTL low at the auxiliary I/O port without changing the sampling frequency, one should refresh the MCR using the following line:

```
MOVEP #$900000,Y:<<MCR
```

In conclusion, one can see that most of the configuration tasks of the DSP56001 processor and the DSP-56 are left to the user of the board, since it is application dependent. Proper configuration is based on a thorough understanding of the hardware and results in a stable and efficient performance of the hardware and software. However, the above discussed configuration tasks are completed at different initialization stages such as booting the system, monitoring and executing user's programs (see the next section for details).

3.2. Hardware Initialization and Program Loading

In the present LiMCA operation, all the three communication ports (Port A, Port B and Port C) of the DSP56001 processor, the 56ADC16 ADC port, which is routed to Port C SSI (Synchronous Serial Interface) for analog input, and the Auxiliary I/O port of DSP-56 board are being used (**Figure 2.7**, **Figure 3.1** and **Figure 3.2**). The configuration of these ports are software controlled and are done by downloading the configuration parameters from the host PC. Thus before all, the Host Port (Port B) of the DSP56001 processor should first be activated in order to set up the communication between the host and the DSP. Then the rest of the DSP ports are configured by the program loaded on the DSP-56.

Before being ready to run user programs, the DSP system first boots itself, establishes the communication to the host processor, and then loads a software monitor. This monitor is controlled by the host and is used to load, monitor and start a user application. The configuration of the system is partially done by the booting process and the monitor software, which includes mainly the configuration of the external memory (Port A) and the host interface (Port B). All the other initialization tasks must be included in the user program.

3.2.1. DSP56001 Booting Process

The booting process is organized in two steps. The first activates the ROM boot strap in the DSP56001 processor. The second loads the DEGMON monitor, which stands for Degenerated Monitor, from the Host PC to the DSP processor.

The DSP56001 processor has four modes of operation controlled internally by bit 0 (MA) and bit 1 (MB) of the Operating Mode Register (OMR) in its program controller, or externally by pin MODA and pin MODB of the processor (**Figure 3.2**). The OMR is a read/write register, thus the mode of the processor is program-controlled. The bit definitions of the register are shown in **Figure 3.13**. The operating modes of the DSP56001 processor are summarized in **Table 3.3**. The DSP-56 board uses mode 1 to boot the processor and mode 2 for user application programs.

After powering on or executing a RESET command, the DSP56001 processor is in the reset state. In this state, the MODA pin and MODB pin are active (**Figure 3.2**). To leave the reset state and start booting, one must apply a high level on the MODA pin and a low level on the MODB pin. When the processor exits the reset state, the two Mode control pins become general purpose interrupt source pins, $\overline{IRQA}$ and $\overline{IRQB}$.

In Mode 1 (Special Bootstrap Mode), A short program saved in ROM (Read Only Memory) is activated. It loads up to 512 24-bit words user's program from the host port (Port B) and save them in the program memory. After the program is loaded, it switches to Mode 2 and transfers control to the user program starting at P:$0000. At this moment, the bootstrap ROM is disabled and shadowed by the program RAM. For details about the other functions and the program listing of the bootstrap ROM see the appendix E of the reference [Motorola 92].

Since in operation mode 1 the bootstrap program can only load a program

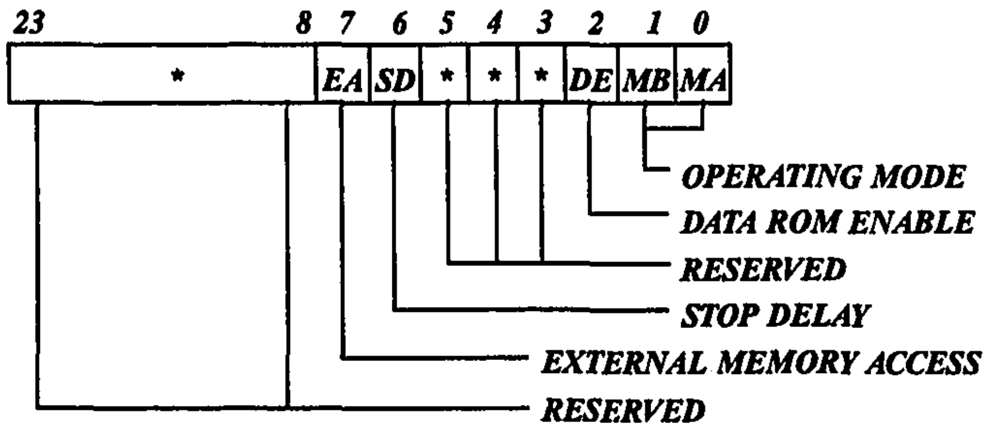


Figure 3.13 Operating Mode Register Format

Table 3.3 Initial DSP56001 Operating Mode Summary [Motorola 89]

Operating Mode	MODB	MODA	Description
0	0	0	Single-Chip Mode
1	**0**	**1**	**Special Bootstrap Mode**
2	**1**	**0**	**Normal Expanded Mode**
3	1	1	Development Mode

smaller than 512 words, most user applications can not be handled at this stage. A monitor, which is less than 512 words, is needed to handle bigger application programs in operation mode 2. This monitor is first loaded to the processor's program memory by the bootstrap program in operation mode 1. Then it takes control of the processor and communicates with the host in Mode 2. Ariel Corp. provided a small monitor program called DEGMON with the DSP-56 hardware. It occupies 64 words of program memory and uses no data memory. Despite its limited number of functions, it is found useful to download a lengthy user application program and then to pass the control of the DSP processor to it. Details about the DEGMON monitor are given in the next section. The source code of the DEGMON.ASM is supplied by Ariel.

The booting process is controlled by the host computer, and three PC's port addresses are used for this. They are:

base + $C000 RESET ON,
base + $8000 RESET OFF,
base + $A000 START BOOTING.

Writing to these ports sequentially invokes the functions listed above. Note that 'base' stands for the base address of the DSP-56 co-processor board. By default, the base address is set to $340, as discussed in **Section 3.1.1**. To change the base address, one must consult the installation procedures in reference [Ariel 89].

The host process downloads the DEGMON monitor through the host port of the DSP processor, while the processor's ROM bootstrap program is running. The monitor must be compiled using the Motorola DSP56000 Macro Assembler and be saved in a file named DEGMON.DAT. The corresponding host protocol, written in Turbo C, is used to carry out the above procedures. During the program downloading in the boot strap mode, polling is used on the DSP side to transfer the program through the host interface. Therefore, the host protocols for program loading is programmed using polling data transfer (**Section 3.1.3.1**). The function prototypes of these protocols can be found in **Appendix A**.

The protocol at the top of the hierarchical structure of the group of the functions is `LoadFile(char *fname, char **result, unsigned int *words, unsigned int *startAddr, int use_mon, int PMemEnable)`. It is a utility to load a DSP process to the DSP processor either in the Special Bootstrap Mode or in the Normal Expanded Mode.

To load the DEGMON monitor, `LoadFile` should be called in the Special Bootstrap Mode by setting the input parameters, `use_mon` and `PMemEnable` to be TRUE, for example `LoadFile("DEGMON.DAT", message, nwords, start_address, TRUE, TRUE)`. In this case, it invokes the function `reset_board(PMemEnable)` to reset the DSP processor and start the booting process.

After successful loading of the monitor, the booting process is completed and a long user DSP application program is ready to be handled by the monitor.

3.2.2. Program Loading through the DEGMON Monitor

On the host side, to load an application program through the DEGMON monitor, the host protocol, `LoadFile` is also used. For example, to load a compiled DSP application named 'LMCDSP.LOD', the following syntax and arguments are used: `LoadFile("LMCDSP.LOD", message nwords, start_address, FALSE, FALSE)`. However, in this case, instead of communicating with the ROM boot strap process, the host interfaces with the DEGMON monitor.

The DEGMON monitor has several sub processes, including an infinite main monitor loop and several host interrupt services. One of them is used to pass the control to a user process (**Figure 3.14**).

After the DEGMON is loaded by the `LoadFile` function, the boot strap process passes control to it, starting at P:$0000, where there is a pointer to jump to the section before entering the main monitor loop. Here, it sets up the external memory (Port A), the host interface (Port B) and the program controller. Then it enters an infinite loop, where it sets up and enables the host interrupts and waits to receive one peripheral data move command opcode followed by an operand from the host and save them at P:$DE_I0 and P:$DE_I1. Finally, DEGMON jumps to the two memory locations starting at P:$DE_I0, executes the opcode, and continues looping.

The opcode and operand are fully controlled by the host. In this way, the host can write to, or read from, the DSP memories, if the opcode and the operand, transmitted from the host, do the data transfer through the HI.

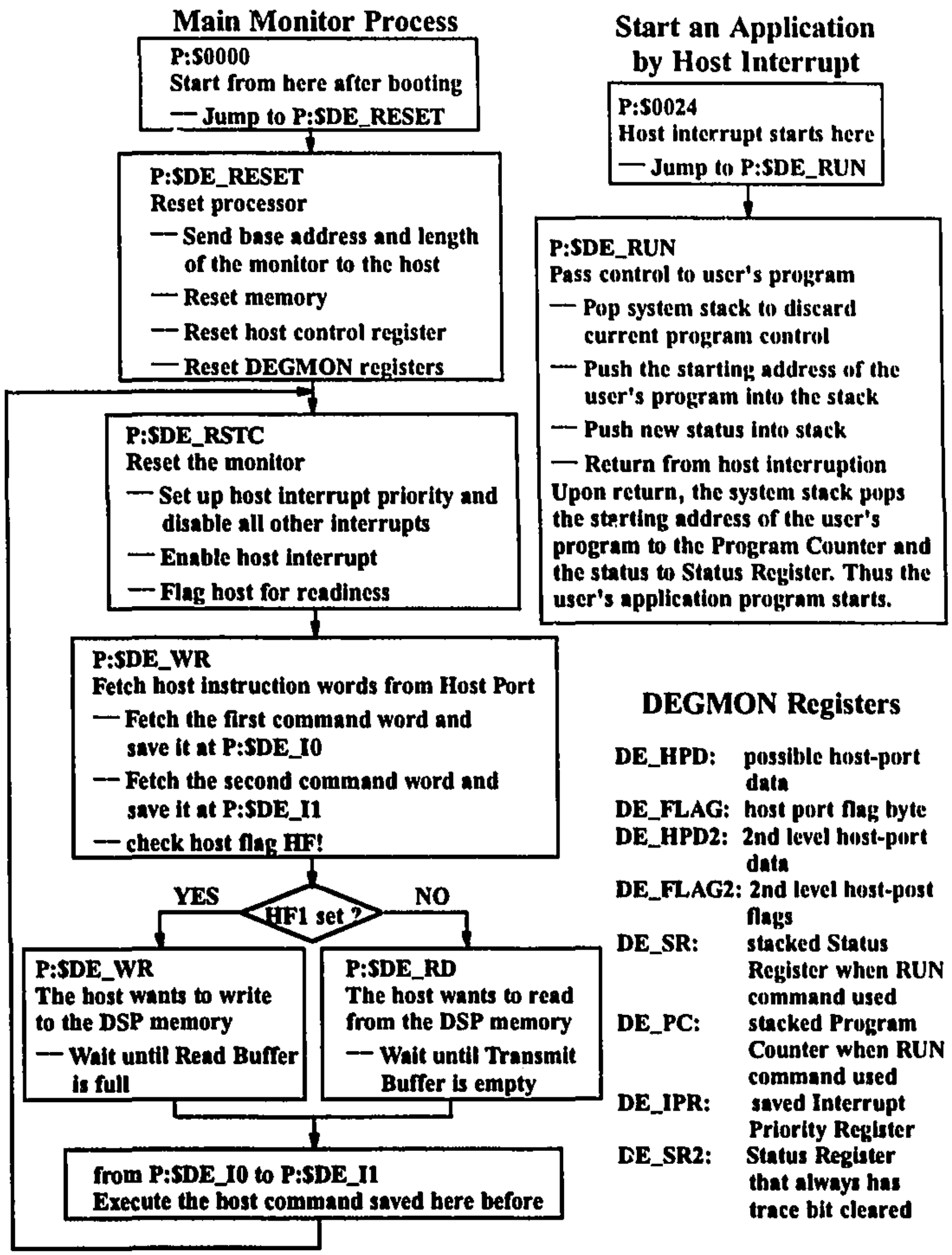


Figure 3.14 Block Diagrams of the DEGMON Monitor

On the DSP side, the opcode needed can be `MOVEP` of both data move directions and of all memory types. For every memory type and data transfer direction, different opcode should be passed to location P:$DE_I0. For instance, to write a word to X memory at X:$1000, the assembly line '`MOVEP x:<<RX,x:$1000`' should be placed at P:$DE_I0 and P:$DE_I1. The corresponding opcode is $0870AB followed by the operand $001000 for the destination address. Therefore, the host must transmit these two words to the DSP process, which saves them at P:$DE_I0 and P:$DE_I1, and then execute them to transmit from the HI to X:$1000. Similarly, to read data from X:$2000, the opcode is $08F0AB, and the operand $2000. There are six different opcodes for bi-directional data transfer for the three types of DSP memories. It is the host's responsibility to choose the right opcode and operand for the data transfer action it wants to. Nevertheless, this is the only way that can guarantee the host to fully control the DSP's data transfer operation.

Furthermore, as discussed in **Section 3.1.3.1**, for different data transfer directions through the host interface, different flags must be polled before the data move. The general purpose host flag HF1 is used by the host to inform the DSP which flag should be polled. If HF1 is set, the host wants to write to the DSP memories, and the HRDF flag in the HSR should be polled by the DSP process. Otherwise, if HF1 is cleared, the HTDE flag should be polled. This indicates that the host wants to read from the DSP memories (**Figure 3.7**). All the above different considerations are implemented by six protocols as:

`readp(unsigned int addr, unsigned long *where);`

`readx(unsigned int addr, unsigned long *where);`

`ready(unsigned int addr, unsigned long *where);`

`writep(unsigned int addr, unsigned long data);`

`writex(unsigned int addr, unsigned long data)` and

`writey(unsigned int addr, unsigned long data)` (**Appendix B**).

These functions are programmed to communicate with the DEGMON monitor, while it is in the main monitor loop.

On the host side, the program loading process consists of a number of function calls to `writep` to download all the opcodes and operands of the user application program to the DSP program memory, and function calls to `writex` and `writey` for any constant variables to be loaded into DSP X and Y data memory.

When program loading completes, the DSP process must be interrupted by the host to break the infinite main monitor loop and to start user application. The host interrupt $24 is used for this purpose. On the DSP side, when the interrupt occurs, the DSP's program controller pushes the present status register and program counter into the system stack and jumps to the interrupt service routine. Upon returning from the interrupt routine, the controller pops up the system stack to restore the status and program counter. To prevent the control from returning to the main monitor loop, in the host interrupt service routine, the system stack is first popped twice to discard the program counter and status before the interruption. Then the starting address and status of the user application program are pushed into the stack. These two pieces of information of the user application were sent to the DSP by the host and saved in the DEGMON registers by the DEGMON monitor through its main monitor loop before the host interruption. As a result, when the interrupt process terminates, the control of the DSP process will be passed to the user application.

A host protocol, `execute_instr(unsigned short startAddr)`, is developed to send the entry data of an application program to DSP and then it invokes `do_host_command(int hc_addr)`, which sends a host interrupt request to the DSP at P:$24 and therefore starts a user application. For details of the host interrupt, see **Section 3.1.3.2.**

As one can conclude, the configuration and initialization of the DSP processor provide a hardware and software platform to run a user application. Most of the configuration and initialization is software controlled. Host protocols have to be developed to customized the hardware and software environment for a specific application. The development and usage of these host utilities are based on a thorough understanding of the DSP hardware and software, which is also essential for the development of the application. The customization of the hardware and software discussed in this chapter is based on the requirement for our LiMCA DSP process, which is detailed in the next chapter.

4. LiMCA SOFTWARE DESIGN AND IMPLEMENTATION

4.1. Software Overview

The software for the DSP LiMCA has been developed based on the hardware described in the previous chapters. It includes the software for the DSP, a host-DSP interface and a Graphical User Interface (GUI). The DSP software performs all the real-time and off-line signal processing tasks. It has been implemented using the Motorola DSP56001 assembly language and runs on the DSP-56 co-processor. The host-DSP interface provides the communication between the host and the DSP board. It downloads the DSP code and configuration parameters to the DSP-56 and starts the DSP processes. During the execution, real-time data are being uploaded from the DSP board to the host through the host interface. The Graphic User Interface eases the job of the LiMCA operators. It provides an "easy-to-navigate" environment with very well organized windows containing input fields, dialog boxes and graphical displays. Furthermore, it performs all the host level computational tasks and controls the DSP processes through the host-DSP interface. The host-DSP interface and the GUI were written in Borland C++. Two commercial software packages, ObjectMenu from Island System and MetaWindow from Metagraphics Software Corporation were used to implement the two interfaces.

In this chapter, the implementation of the DSP software and host-DSP interface will be discussed. Special attentions are paid to the real-time DSP processes. The GUI is not in the scope of this thesis. For details, see [Draganovici 94].

4.2. DSP Software

The DSP software was designed as a group of real-time and off-line tasks. The selection and execution of any one of them is controlled by the host computer through the host interface. A command interpreter has been developed to monitor the host interface and to pass the control of the processor to the appropriate task. The logical structure of the command interpreter is illustrated in **Figure 4.1**. Its source code is found in Appendix C, starting under the label '`CMDLUP`' on page 112. The function pointer table in **Figure 4.1** is located in internal X RAM (see the code under the variable '`flist`' on page 96 in Appendix C).

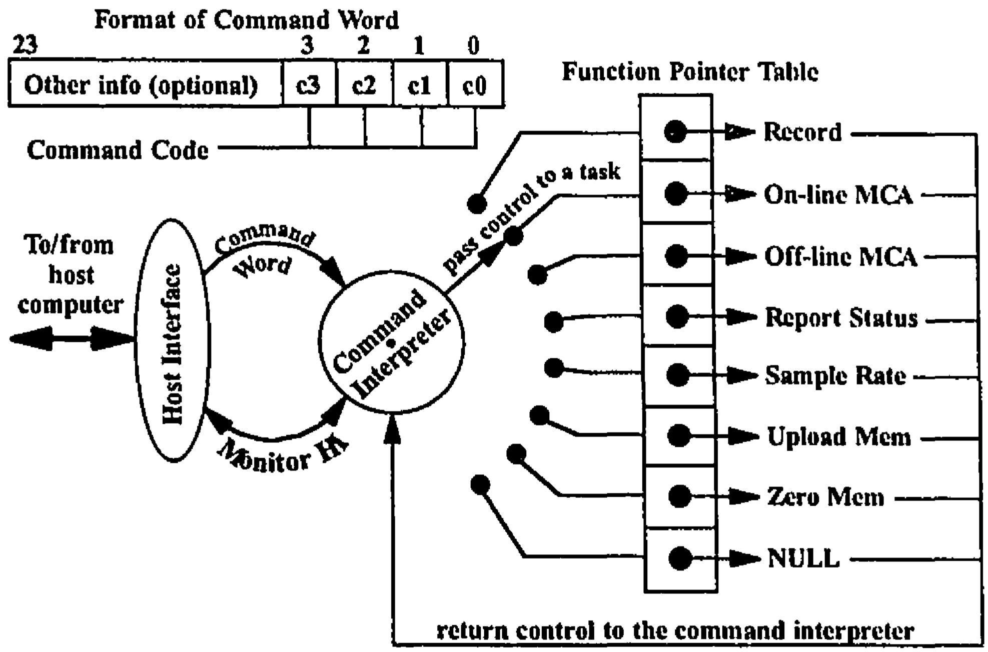


Figure 4.1 Format of the Command Word and Logic of the Command Interpreter

The entry point of the DSP software is at the label 'INIT_PGM' (see page 111 of Appendix C). After being downloaded by DEGMON monitor and taking over the control of the DSP processor, the DSP software first initializes the SSI interface for ADC and sets up ADC sampling frequency (for DEGMON monitor and program loading, see **Section 3.2**, and for SSI and sampling frequency setup, see **Section 3.1.4 and 3.1.5**). Then it enters the command interpreter, waiting for a command word from the host computer. The format of the command word is also shown in **Figure 4.1**. The least significant nibble is used to carry the command code, that tells the interpreter to which task to pass the control. For the four-bit command code used, a maximum of 16 tasks can be managed. Presently 7 slots are used, and the rest provide space for further development. The other bits of the command word are optionally used for additional information needed by certain processes. For example, to start the real-time multi-channel analysis (MCA), the host uses bit 4 and bit 3 to inform the DSP which analog channel should be used or if both are to be used. Bit 4 is for channel A and bit 3 for channel B. Toggling any one of the two bits enables, if it is set, or disable, if it is cleared, the channel which it represents.

4.3. DSP Real-time Software

Among the selections listed in **Figure 4.1**, the real-time LiMCA process is carried out by the On-line MCA. This is organized as a number of independent tasks, each designed as a filter, reading data from an input buffer and writing new data into an output buffer. **Figure 4.2** shows these tasks together with the corresponding data flow paths. A small real-time control executive, which is not shown in this figure, was developed to manage their execution. It receives a number of parameters from the host, and the starts the execution of the different DSP tasks according to the status of the buffers and processes involved. Note that **Figure 4.2** 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 (one per channel). The circular buffer is processed by the *peak sampling* process that detects the presence of peaks and transfers peak data to the 'sampled peak buffers'. A digital filter can be invoked before the *peak sampling* process to eliminate noise (say, from an induction furnace near by). The 'sampled peak buffers' are processed by the *peak description* process which stores its output into the 'peak buffers'. The *pulse height analysis* process processes and modifies this information which is then passed to the host through the 24-bit host port. At the host

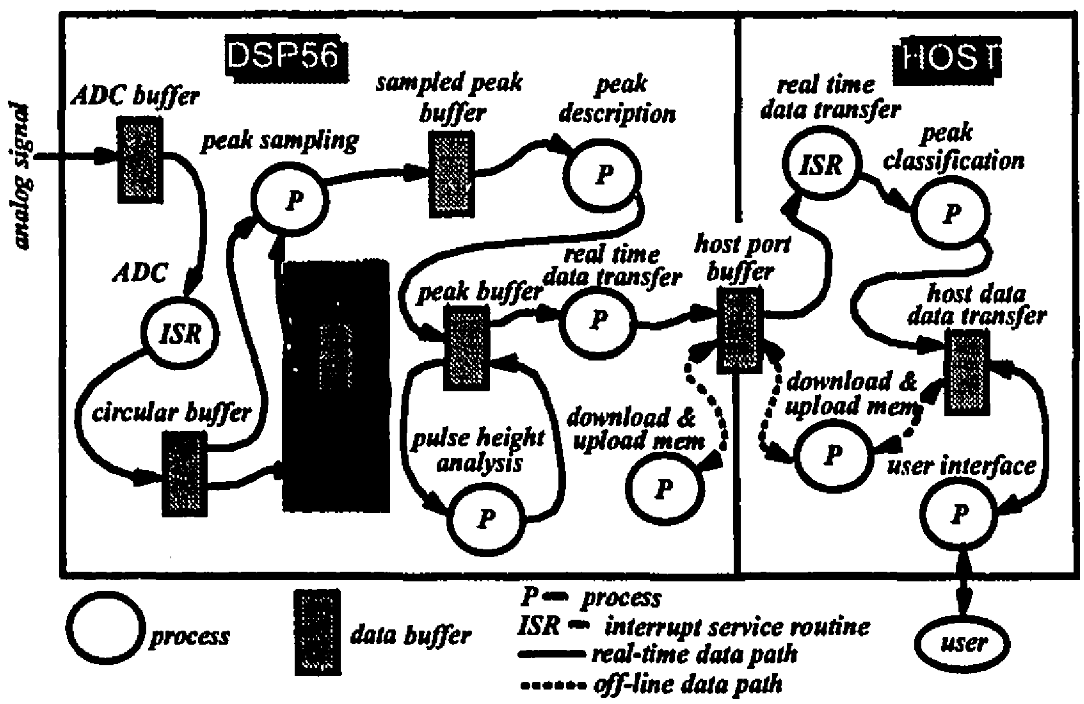


Figure 4.2 The Structure of the DSP Real-time Software

level, an ISR is invoked to read the data from the host port and pass them to the peak classification task.

4.3.1. Task Distribution between the Host and DSP

The real-time processes shown in **Figure 4.2** were first distributed between the DSP-56 co-processor board and the host computer, and then coded separately. The scope of this task allocation is to take full advantage of the DSP pipeline architecture and to maximize its utilization. Time is 'wasted' when the DSP co-processor board communicates with the host. Due to the pipeline architecture of the Motorola DSP56001 processor, two types of operations are most time consuming, control transfer instructions and instructions that perform data transfers between the DSP-56 and the host. Along the data path illustrated in **Figure 4.2**, the whole process can be viewed as a data reduction process in terms of the amount of data processed in each sub process from ADC to peak classification. To minimize the data needed to be transferred through HI, the processes that do major data reductions are required to be programmed at the DSP level. Such processes include *ADC*, *peak sampling* and *peak description* processes. As a result, the bulk of the data transferred between the DSP processor and the host consist of the peak description parameters.

It was also intended to code the *peak classification* process at the DSP level, thus the host can be freed from low-level data-intensive processing and ultimately be used for high-level operation such as GUI and calculations with sophisticated algorithms. However, in the initial prototype of the DSP-based LiMCA system, we decided to implement a prototype of the peak classification process at the host computer level. This decision was influenced by the fact that the system is being developed for a research environment and will be used in different melts and under different conditions. By coding the peak classification at the host level we increase the ease with which the code can be enhanced to accommodate different situations. In addition, we plan to investigate the use of fuzzy logic and artificial neural networks for this task, and this is easier at the host level.

4.3.2. Memory Allocation at the DSP Level

Due to the hardware architecture of the DSP56001 processor, the allocation of the DSP memory greatly affects the efficiency of the DSP performance. Such memory distribution includes the allocation of the DSP program memory to the DSP processes and the DSP X and Y data memories to the buffers, variables, stacks, status and control registers.

As one can see in **Figure 3.5**, the DSP56001 has 512 Kwords of internal program RAM and the same number of RAM words for the X and Y data memories. Each bank of the internal RAM is accessed through its own data and address buses, thus several banks can be accessed in parallel in one instruction cycle. However, the majority of the RAM used for program and data is external memory, physically implemented on the DSP-56 board. All of the external RAM is accessed via the communication Port A of the DSP56001 processor, i.e. the external program RAM, X and Y data RAM share the same data and address buses (**Figure 3.1**). As a result, parallel data move is not applied to the external memories. Bit operation instructions and jump instructions on bit status are also not applicable. Furthermore, instructions saved in the external program RAM space may break the instruction pipe line, thus introduce extra delays.

The DSP56001 instructions are so pipelined during execution that the DSP CPU fetches an instruction from the program memory, decodes the instruction previously fetched and executes the instruction previously decoded, all in one instruction cycle. If the execution of the instruction involves a data move to or from any of the external RAM locations and the instruction to be fetched resides in the external P RAM, both actions require the use of the external data and address buses, therefore they can not be

Table 4.1 The Usage of the Program Memory

Starting Address	Process	Length (words)
P:$0000	Interrupt Vector Space	64
P:$0040	Degmon Monitor	80
P:$0090	ADC Interrupt Service Routine	20
P:$00A4	Host Stop Interrupt Service Routine	5
P:$00A9	Peak Sampling Process	223
P:$0188	Peak Description Process	137
P:$0211	Pulse High Analysis Process	28
P:$022D	DSP to Host Data Transfer for Channel A	19
P:$0240	DSP to Host Data Transfer for Channel B	20
P:$0254	Control Executive Process	106
P:$02BE	System Initiation	34
P:$02E0	Utilities	209

Table 4.2 The Allocations of X and Y-data Memories

X-RAM		Y-RAM	
Address Range	Purpose	Address Range	Purpose
X:$C000-$0001	global registers	Y:$0000-$0001	global variable
X:$0002-$0009	variables & registers for Channel A	Y:$0001-$0009	variables & registers for Channel B
X:$000A-$0011	global variables	Y:$000A-$000B	stacks
X:$0012-$0019	pointer table to functions		
X:$001A-$001E	circular buffer pointers, time counters		
		Y:$0100-$04FF	PHA reference table
X:$6000-$65FF	peak parameter buffer for channel A	Y:$6000-$65FF	peak parameter buffer for channel B
X:$7000-$7FFF	sampled peak buffer for channel A	Y:$7000-$7FFF	sampled peak buffer for channel B
X:$8000-$FBFF	circular buffer for channel A	Y:$8000-$FBFF	circular buffer for channel B

completed in parallel. Thus the pipeline must be broken to avoid bus conflict.

For best efficiency, all the effects of the architecture of the DSP-56 processor must be considered when the memory utilization is planned. In our application, the most data-intensive DSP processes are placed in the internal program memory. From the discussion in **Section 2.3.2**, the busiest processes are the *ADC* and the *peak sampling* processes, which are loaded into the internal low program memory space. Off-line subroutines are placed into the high external memory space. The usage of the program RAM by the DSP processes are listed in **Table 4.1**. For complete DSP source code listing of the DSP LiMCA, see Appendix C.

The X and Y data memories are used for buffers, variables, stacks, status and control registers of various DSP processes. The internal data memories should be reserved for variables, stacks, status and control registers. As mentioned before, bit operation instructions and jump instructions conditioned on bit status can only apply to the internal memories. The former type of instructions are frequently used to manipulate status and control registers and the latter are used to direct the process properly according to the status of the bit flags of the status and control registers. Thus, these registers are required to be located in the internal memories. The variables and stacks used by data-intensive real-time processes should also be put into the internal memory to maximize parallel data movement. Buffers, which are usually too big to be put into the internal memories are placed into the external data memories. For the LiMCA process, the X data memory is primarily allocated to the buffers for the analog-channel-A DSP processes and the Y data memory to the buffers for the analog-channel-B DSP processes. **Table 4.2** shows the allocation of the X and Y data memories.

Two 31-Kword circular buffers are located at the bottom of the X and Y data memories, for the digitized data from analog channel A and channel B respectively. These buffers increase the time elasticity of the real-time DSP process, important especially when the worst case of operation frequently occurs (see **Section 2.3.1** for the worst case of operation).

4.3.3. Real-time Control Executive

The overall real-time DSP task is being conducted by invoking, based on certain conditions, one of the sub-processes at a time. There are several real-time sub-processes involved, namely *ADC*, *Peak Sampling*, *Peak Description*, *Real-time Data Transfer* (**Figure 4.2**). Each has its own entry conditions. Considering a two-channel system, the conditions of the signals from the two analog channels are different at most of the time. The number of the sub-processes and the complex entry conditions of these processes complicate the program coding and maintenance. An executive process is needed for control and has been developed to monitor and orchestrate the real-time sub-processes, thus we can modularize the program coding and ease the program debugging and maintenance.

The communication links between the control executive and the host computer and the real-time sub-processes are schematically illustrated in **Figure 4.3.** Note that in this figure, the process *Transfer A* transfers channel-A 'peak buffer' to the host and *Transfer B* transfers channel-B 'peak buffer' to the host. The control executive has two states of operation, the initialization state and the real-time monitoring state. **Figure 4.3** depicts the communication links while the executive is in the monitoring state.

The executive is activated by the command interpreter, when it passes the control to the function labeled with '`onlineMCA`' (**Figure 4.1**). After taking control, the

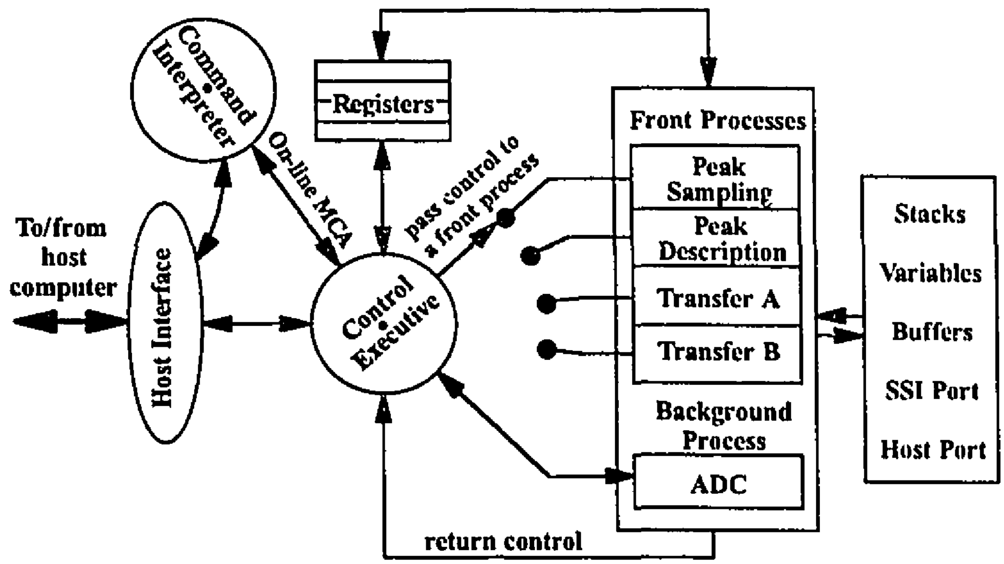


Figure 4.3 Real-time Control Executive and its Communication Links

executive first enters the initialization state to set up all the needed registers, stacks, variables, buffers, communication ports and the pointers to the background functions servicing the ports. In this state, it also communicates with the host computer to receive process parameters. Such parameters include noise thresholds and processing time for the *peak sampling* process (**Section 4.3.5**), PHA channel number, PHA checking table and PHA quick sort cycle number for *PHA* process (**Section 4.3.7**). Then it signals the host for the readiness of conducting real-time DSP and waits for a host response code. The host can either send a 0, to start the real-time process or a 1, to quit. In the latter case, the executive immediately returns control back to the command interpreter.

A zero from the host at this stage changes the executive state from the initialization state into the real-time monitoring state. Upon entering this state, the executive starts the background process, i.e. *ADC* process, enables the host interrupt so that the host can terminate the process at any time by sending a host interrupt through the host port. In this state, it monitors the flags in the registers which reflect the conditions of the buffers and processes involved. Depending on the status of the flags, it selects and activates the required process at the proper timing.

The registers used here are shown in **Figure 4.4**. Note that the undefined bits are not used. The ProcStatus register reflects the status of the real-time MCA process. Bit 0 of the register, the HOSTSTOP flag, indicates if the host has instructed the DSP to stop the process by the host command interrupt $24 (**Section 3.1.3.2**). It is set when

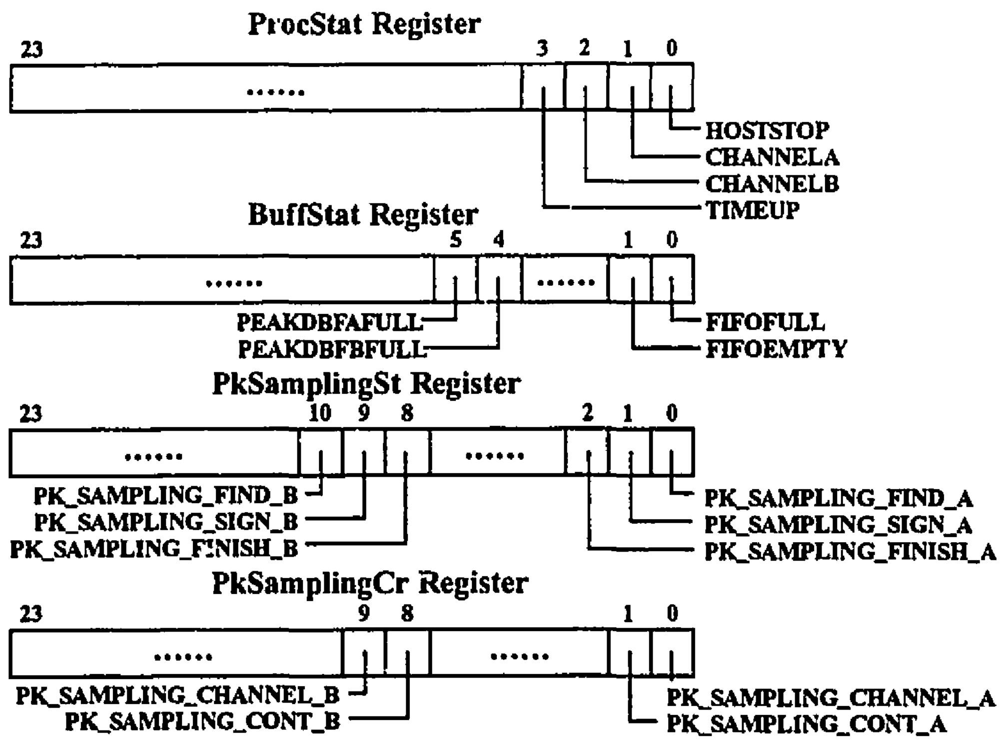


Figure 4.4 Registers of the Real-time MCA Process

the host wants to terminate the process. Bit 1 and 2, CHANNELA and CHANNELB, are analog channel flags. CHANNELA is for channel A and CHANNELB for channel B. They are set when the corresponding channels are enabled. They reflect the mode of the ADC operation, i.e. one-channel mode or two-channel mode (stereo mode). These two flags are initialized during the initialization stage and are not changed later. This indicates that the ADC operating mode can not be changed during real-time processing. Bit 3, the TIMEUP flag, is set when the processing time exceeds the time which is pre-set by the host.

The BuffStat register flags the status of some buffers used in the real-time processing. Bit 0, i.e. the FIFOFULL bit, is used as the buffer full flag for the two circular buffers. It becomes set when both of the buffers are full. This indicates a buffer overflow error, which causes the real-time process to fail. Bit 1, i.e. the FIFOEMPTY flag, is used for the executive to monitor whether the circular buffers are empty after the process is commanded to terminate for any reasons and the circular buffers have to be emptied so as not to lose data after the termination. It is set when either the HOSTSTOP flag or the TIMEUP flag in the ProcStat register is set and the circular buffers are empty in normal termination conditions. Such conditions occur when the host terminates the DSP process or the processing time exceeds the pre-set time limit. The FIFOEMPTY flag is also set after FIFOFULL is set and the circular buffers have been flushed. This is an abnormal termination. The FIFOEMPTY flag provides the exit condition for the control executive. In any case, when this flag is set, the executive signals the completion of the process to the host by setting Host Flag 2 (HF2 in **Figure 3.7** and **Figure 3.8**) and reports the exit conditions to the host by sending both the ProcStat and BuffStat registers to the host port and returns the control back to the command interpreter.

Bits 4 and 5 of the BuffStat register, i.e. the PEAKDBFAFULL and PEAKDBFBFULL flags, are used to flag the status of the 'peak buffers' (**Figure 4.2**) for channel A and channel B respectively. PEAKDBFAFULL is for channel A and PEAKDBFBFULL is for channel B. When either of the 'peak buffers' is full, the corresponding flag becomes set. This indicates that the real-time process can not continue unless the buffer is flushed and thus available for new peak parameters. Upon seeing the buffer full flags set, the executive immediately passes control to one of the two real-time data transfer processes depending on which buffer is full and should be transferred to the host. If PEAKDBFAFULL is set, *Transfer A* is called or otherwise *Transfer B* is called.

The executive normally passes control to the *peak sampling* and *peak description* processes in sequences, unless one of the flags in the BuffStat becomes active, signaling the need for urgent attention and immediate action.

The PkSamplingSt and PkSamplingCr registers are mainly used by the *peak sampling* and *peak description* processes. Their explanations are included in the descriptions of the two processes in later sections. The source code of the real-time control executive is listed in Appendix C starting at '`OnlineMCA`' on page 109.

4.3.4. ADC Process

The *Analog-to-Digital* (*ADC*) conversion process has been implemented as a background interrupt-driven process. It uses the SSI Receive Data Interrupts, which are located at P:$000C and P:$000E of the Interrupt Vector Table in the program RAM space (**Table 3.1**). To start the process, Port C and the interrupt priority level have to be configured properly. The entry pointer to the ADC interrupt service routine must also be installed at the two vector spaces mentioned above. If the sampling rate required is different from the default 50 KHz, it should be set using the Sample Rate

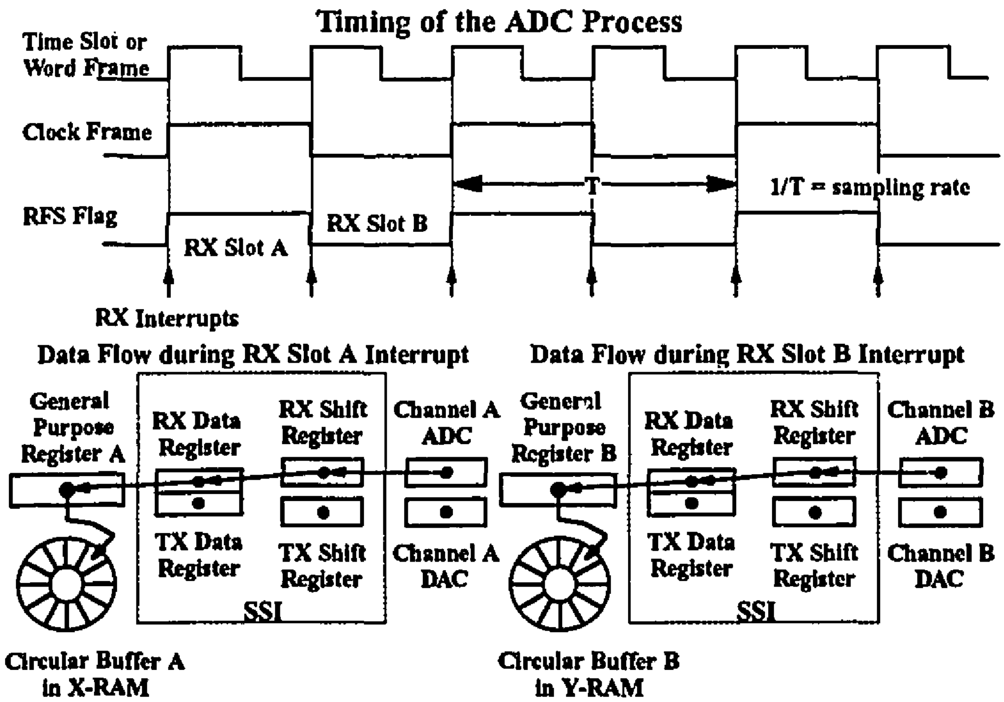


Figure 4.5 The Timing Diagram and Data Flow of the ADC Process

function through the host interface and the DSP command interpreter (**Figure 4.1**). Details about the setups and configurations are given in **Section 3.1.4** and **3.1.5**. The source code listing can be found in Appendix C starting at '`ssIDataInPtr`' on page 97. Note that a different approach was used to install the entry pointer of the *ADC* process into the SSI Interrupt Vector spaces from the one using ORG directive, explained in **Section 3.1.4**. Here a utility function, '`InstallssIInts`' is used to install the interrupt service routine (see page 114 in Appendix C).

The *ADC* process is invoked by the real-time control executive when it enters the real-time monitoring state. The process reads data from the ADC buffer at X:$FFEF and saves them into the circular buffers for both channel A and B. Since the data for both channels are from the same register in SSI, they are differentiated by different timing, and there is a delay of one time slot between the two channels, see the timing diagram in **Figure 4.5**.

Compared with the timing diagram in **Figure 3.12**, The *ADC* process, that we used here for the LiMCA application, only services the ADC part of the SSI interrupts. Nevertheless, it manages two circular buffers, shown in **Figure 4.6**. Note that the two circular buffers are managed by one set of pointers. This indicates that the *ADC* process

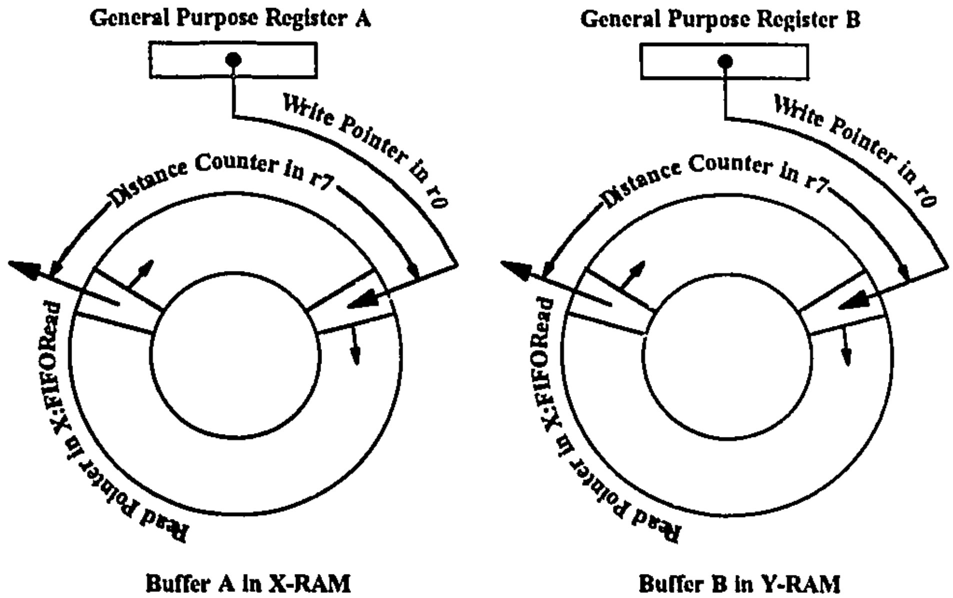


Figure 4.6 Circular Buffers for ADC

does not know which analog channel is in use. It is always working in stereo mode. The buffer read pointer is saved in the address register r0, which is one of the eight address registers from (r0 to r7). This register is not stacked when the processor switches between the background and foreground processes. Thus it should not be used by the foreground processes. Address register r7 is used as a counter, counting the distance between the write pointer and the read pointer. It is transparent between the background and foreground processes. When data are written to the circular buffers, the *ADC* process increments r7. When data are read from the buffers, the *peak sampling* process decrements r7. The BUFFER FULL condition is checked by the *ADC* process. If r7 equals the size of the circular buffers, the buffers are full and FIFOFULL flag in BuffStat register is set and the process is terminated by disabling the SSI interrupts. The BUFFER EMPTY condition, i.e. r7 equals 0, is monitored by the foreground process, the *peak sampling* process, which reads data from the circular buffers.

All the measures discussed above simplified the programming of the *ADC* process and therefore increased the efficiency of the real-time process.

4.3.5. Peak Sampling Process

As discussed in the previous section, the digitized LiMCA signal is saved in two 31-Kword circular buffers, one per channel. The real-time analysis of the signal is further carried out by several foreground processes under the control of the real-time control executive. The analysis task is decomposed into *peak sampling*, *peak description* and *peak classification* processes, **Section 1.4.**

Here the *peak sampling* process finds and chops the peaks in the circular buffers based on two noise thresholds, which mark the margins of the noise band of the signal (**Figure 4.7** and **Figure 4.8**). The noise band reflects the on-spot operational conditions and determines the minimum size of the particles that the system can detect under such conditions. The process also manages the read pointer of the two circular buffers, monitors the buffer empty condition and updates the processing time when it reads the circular buffers.

The LiMCA signal, as shown in **Figure 4.7**, has two states: State 1 (no peak state) indicates that the digitized data are within the noise band and State 2 (peak state) indicates that the data points are beyond the noise thresholds.

The flags in PkSamplingSt register mark the state of the signal, sign of the peak being sampled and the completion of the peak sampling for both analog channels (**Figure 4.4**). The PK_SAMPLING_FIND_A or PK_SAMPLING_FIND_B flags are used to reflect the state of the signal from channel A or channel B, zero for State 1 and one for State 2. In State 2, a peak is being sampled and the sign of the peak is marked by PK_SAMPLING_SIGN_A or PK_SAMPLING_SIGN_B, 1 for positive peak and 0 for negative peak. The PK_SAMPLING_FINISH_A and PK_SAMPLING_FINISH_B flags are used to mark the completion the peak sampling. Upon completing the sampling of a peak for one channel, the completion flag for that channel becomes set.

As shown in **Figure 4.8**, a peak is sampled started at the point right before the one that crosses one of the noise threshold and ended at the first point restoring back into the noise band. Positive and negative peaks are sampled in the same way. The sampled peak is saved in the 'sampled peak buffers' (**Figure 4.2**).

Accompanying with the sampled peak, the width of the peak is saved at X:PkWidthA if the peak is from channel A or at Y:PkWidthB if the peak is from channel B.

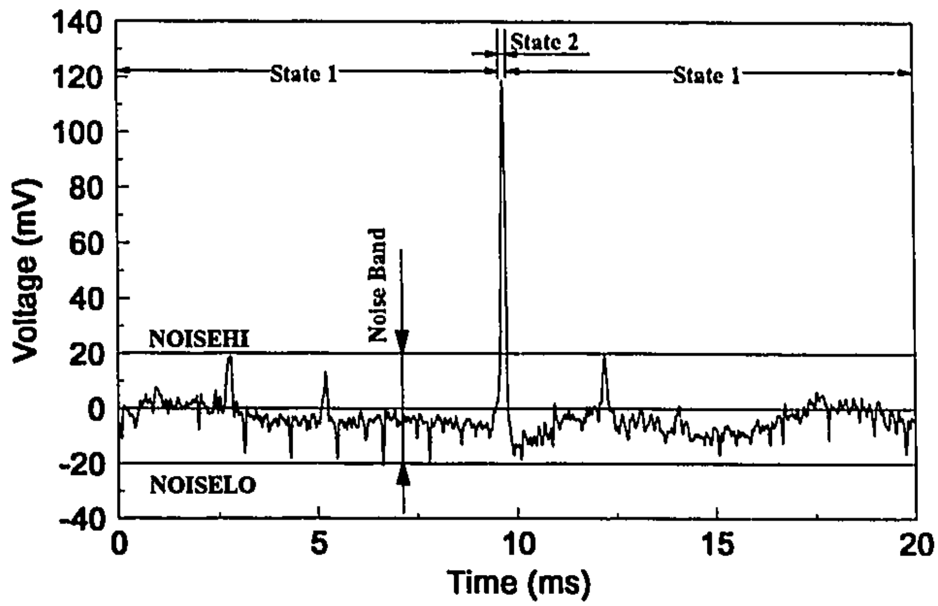


Figure 4.7 A Typical Section of LiMCA Signal Extracted from the Eastalco Aluminum Test

The time label of the first data of the peak is also saved. It is used later to compute the time when the peak starts. Two variables are used to form a 32 bit counter for the time label. Therefore only the lower 16 bits of the two variables are used. X:PkStartLo16A keeps the low 16 bits of the counter and X:PkStartHi16A keeps the

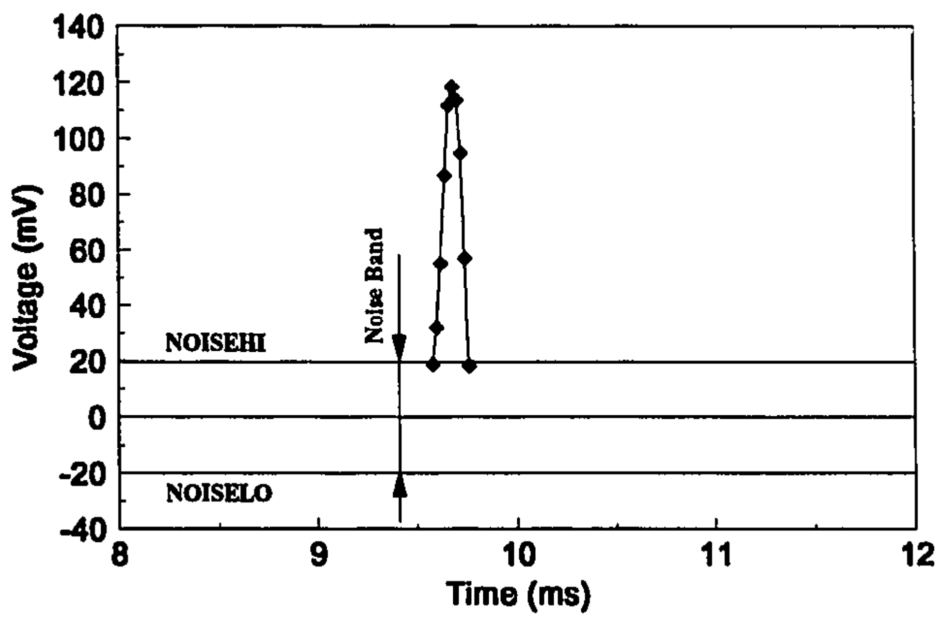


Figure 4.8 The Peak Sampled from the Signal in Figure 4.7

high 16 bits of the counter for the peak from channel A. Similarly, Y:PkStartLo16B and Y:PkStartHi16B keep the low and high 16 bits of the counter respectively for the peak from channel B.

Depending on the state of the signal at the time that the process starts or terminates, there are two entry or exit conditions. They are the 'COMPLETE' and 'CONTINUE' conditions, which are represented by the PK_SAMPLING_CONT_A flag for channel A and the PK_SAMPLING_CONT_B flag for channel B in the PkSamplingCr register (**Figure 4.4**). The flags are set to 1 for 'CONTINUE' condition and 0 for the 'COMPLETE' condition. The states of the two flags are validated before the process exits. If the process returns when the signal in one channel is in state 1, the corresponding flag is cleared to indicate the 'COMPLETE' condition. Otherwise, the flag is set to indicate the 'CONTINUE' condition. This condition indicates that the sampling of the current peak has been interrupted and must be resumed later.

The next time the process is invoked, it decides either to continue sampling the peak that was not finished before it exited or to find a new peak according to the conditions of these two flags.

In order to sample the peaks in the circular buffers in real-time, the *peak sampling* process must run at the same pace as the *ADC* process, which is writing the data into the circular buffers. The speed of the *peak sampling* process is controlled by monitoring the circular buffer empty condition. As shown in **Figure 4.6**, the read pointer managed by the *peak sampling* process is always "chasing" the write pointer managed by the *ADC* process. The distance between the two pointers is monitored by the r7 address register. Each time the *peak sampling* process fetches data from the two circular buffers, it checks if r7 is zero (BUFFER EMPTY). If it does turn out to be zero, the buffers are empty, there are two possible cases: the buffers are temporary empty, which frequently occurs since generally the *peak sampling* process is faster than the *ADC* process (by the design of the software to avoid the buffer overflow condition). In this case, the *peak sampling* process introduces idle cycles to wait for the *ADC* process to fill the circular buffers and then resumes processing when the buffers are not empty. In this way, the read pointer is prevented from outpacing the write pointer, and thus the background process and foreground process are kept at the same pace.

The second possible case is when the *ADC* process has been terminated and thus stopped filling the circular buffers. In this case, the HOSTSTOP flag in the ProcStat register became active for the *peak sampling* process to poll. Under this condition, the process sets the FIFOEMPTY flag in the BuffStat register and returns. As discussed in

Section 4.3.3, this flag will cause the control executive to do the necessary clean-up and terminate the real-time MCA process.

The processing time is recorded and updated each time when the process reads a new set of data from the two circular buffers. The time is counted in a 32-bit counter formed by two variables at X:CountLo16 and X:CountHi16 for the low and high 16 bits of the counter respectively. This counter is compared, each time it is incremented, with the pre-set maximum processing time saved at X:CountLo16Max and X:CountHi16Max. If the two 32 bit values are equal, the process sets the TIMEUP flag in the ProcStat register, stops the SSI interrupts and sets r7 to zero, making the circular buffers empty. These actions cause the control executive to ignore the digitized data in the circular buffers that came later than the data currently processed by the *peak sampling* process, and therefore to terminate the real-time process immediately at the time that the host has expected. The default time counts at X:CountLo16Max and X:CountHi16Max are $00FFFF, or otherwise specified by the host computer and downloaded to the DSP process. The default values of the time counts represent a time span of about 23.86 hours at the sampling frequency of 50 KHz. This is clearly an unrealistic processing duration. However this situation is frequently used to allow the user to monitor the progress of the processing and to terminate the process at any time using the host command interrupt.

In conclusion, the *peak sampling* process samples the peaks from the circular buffers and detects the exit conditions for the real-time control executive. The source code for this process can be found in Appendix C starting at '`PkSampling`' on page 98.

4.3.6. Peak Description Process

After a peak from either channel A or channel B is sampled by the *peak sampling* process, the *peak description* process is invoked by the real-time control executive. The entry condition for the process is defined in the PkSamplingSt register. The PK_SAMPLING_FINISH_A and PK_SAMPLING_FINISH_B flags are checked in order to decide which input buffer ('sampled peak buffer A' or 'sampled peak buffer B'), and which output buffer ('peak buffer A' or 'peak buffer B') to be used.

This process analyzes data from the 'sampled peak buffer' and generates a six parameter description of each peak. These include four shape and two time parameters (**Figure 4.9**). The shape parameters are the *peak height*, the *width*, the *start slope* and the *end slope*, and the time parameters are the *start time* and the *peak time*.

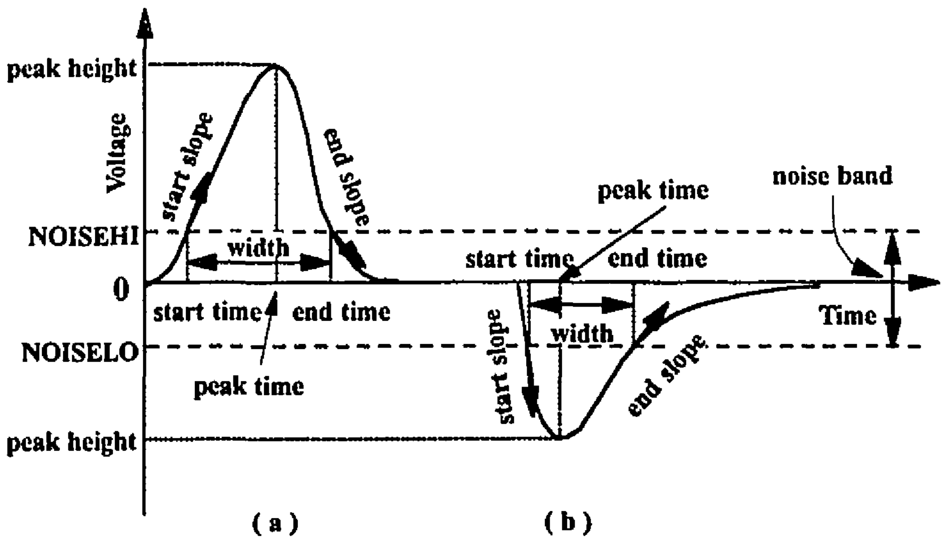


Figure 4.9 Peak Parameters: (a) Positive Peak, (b) Negative Peak

The *start time* has been already counted by the *peak sampling* process and saved in the variable pairs, X:PkStartHi16A and X:PkStartLo16A for channel A or Y:PkStartHi16B and Y:PkStartLo16B for channel B, which represent the absolute time in the format discussed in the previous section. The *width* of the peak has also been computed by the *peak sampling* process. It is saved in X:PkWidthA for channel A or Y:PkWidthB for channel B.

Other parameters are to be calculated by this process. The *start slope* and *end slope* are the derivatives of the peak at the *start time* and *end time* respectively. The *start slope* is calculated by subtracting the second data point from the first data point of the 'sampled peak buffer' and the *end slope* by subtracting the last data point from the second-to-last data point in the same buffer. The *peak time* is defined as the time when the data point reaches the positive or negative maximum point for a positive or negative peak respectively. It is computed as the data count from the *start time* to the *peak time*. Thus it is a relative time label. The peak height is found by comparison and is represented by 16-bit signed integer number corresponding to the 16 bit ADC interface.

After the computation of the above parameters, the process calls the *PHA* process to find the *PHA channel number* associated with the height of the current peak. If the peak is negative, a zero is returned, otherwise the related *PHA channel number* is returned (see next section). This and the six peak parameters form a group of seven parameters in total, characterizing a peak in the time domain. They are transferred into

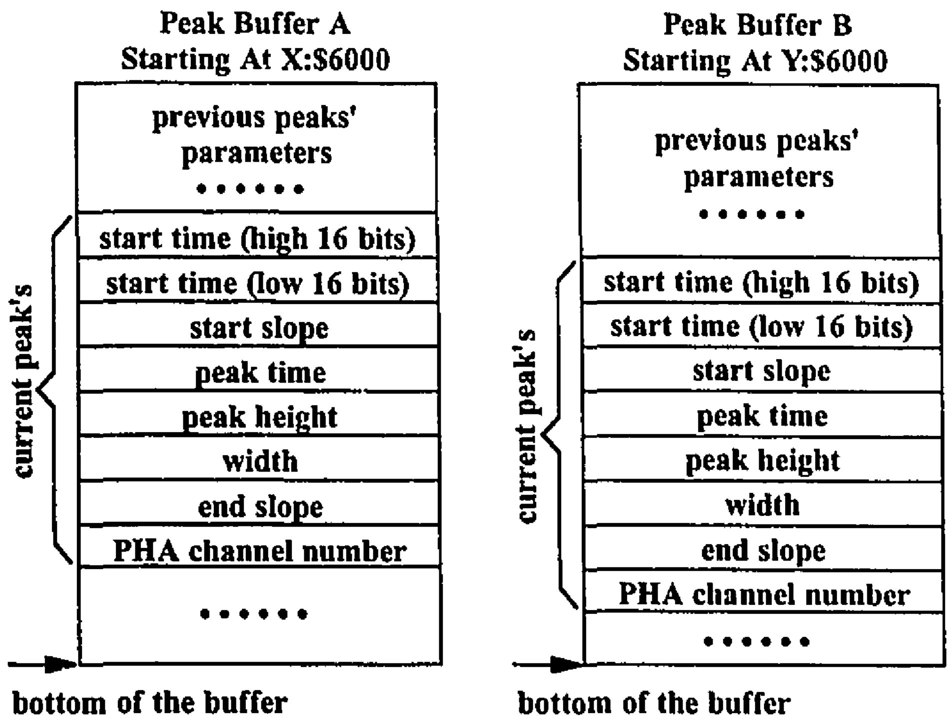


Figure 4.10 Parameter Sequence in the Peak Buffers

the 'peak buffer' (**Figure 4.2**) before the process returns. They occupy eight memory locations (the *start time* takes two locations). The sequence of the parameters in the 'peak buffer' is shown in **Figure 4.10**.

The conditions of the two 'peak buffers' are checked by this process. If any of them is full, the related buffer full flag in the BuffStat is set to signal the control executive to transfer the buffer to the host and empty the buffer by calling a data transfer utility, see **Section 4.3.8**.

In brief, the *peak description* process describes the peaks saved in the 'sampled peak buffers', saves the peak parameters in the 'peak buffers' and manages the 'peak buffers'. Its source code is listed in Appendix C under the label '`PkDescription`' on page 103.

4.3.7. Pulse Height Analysis (PHA) Process

The *PHA* process takes the height of a positive peak, calculates and returns the PHA channel number that corresponds to the peak height. To emulate the operation of the analog LiMCA system shown in **Figure 1.5**, a digital logarithmic amplifier must be implemented. Since the Motorola DSP56001 processor, a fixed point processor, is being used, to avoid floating point calculations, a table-driven algorithm is used. Suppose the height of a peak is y, which is in the range of

$$y_{min} \leq y \leq y_{max},$$

here y_{min} can be the height of the smallest peak that can be detected and y_{max} is the up limit of the ADC. In our case, a 16-bit ADC is used and the digitized number is represented in a signed binary format. The number range is from -32768 to +32767. Thus here $y_{max} = 32767$.

To emulated the log amplifier, take

$$Y_{min} = LOG(y_{min}),\ Y = LOG(y) \text{ and } Y_{max} = LOG(y_{max}) \qquad (4.1)$$

then

$$Y_{min} \leq Y \leq Y_{max}.$$

To find the PHA channel number, Y is compared with the series:

$$Y_i = i \cdot \Delta Y, \qquad 0 \leq i \leq N-1 \qquad (4.2)$$

where N is the total number of the PHA channels and

$$\Delta Y = (Y_{max} - Y_{min}) / N.$$

For a certain integer k, if

$$Y_k \leq Y < Y_{k+1} \qquad (4.3)$$

then k is the channel number that corresponds the peak of height y before the logarithmic amplification.

To avoid using **Equation (4.1)** in processing, we transform the series **(4.2)** into

$$y_i = EXP(Y_i) \qquad (4.4)$$

Considering **(4.1)** and **(4.4)**, **Equation (4.3)** is equivalent to

$$y_k \leq y < y_{k+1} \qquad (4.5)$$

Equation (4.5) is used in real-time processing. A exponential PHA checking table is constructed using Equation **(4.4)** and is located in Y data memory space starting at Y:$0100. The length of the checking table **N,** being 256, 512 or 1024, equals the total number of the PHA channels. The contents of the exponential checking table are computed by the host and then downloaded to the DSP. The following piece of code on the host side is used:

```
(float)ChannelInc=log(32767)/ChannelNum
for(i=0; i<ChannelNum; i++)      {
    (int)PHATable[..]=pow(10.0,i*ChannelInc)
    }
```

The variables are:

`ChannelInc`: logarithmic increment between two adjacent channels;

`ChannelNum`: total number of PHA channels;

`PHATable[ ]`: an array to keep the contents of the PHA checking table.

Note that in this sample program y_{min} is assumed to be 1 and y_{max} to be 32767. Here only positive peaks are concerned. However, negative peaks can be handled in the same manner after being negated.

A quick sort routine compares the peak height with the contents of the table to find the two consecutive values, which satisfy **equation 4.5 (Figure 4.11)**. The address of the memory location which keeps the lower value of the two is used as an index to the PHA channel number. The channel number is later computed by subtracting the base address of the table from the index. The number of the comparisons of the process, also called the number of sorting cycles, is a constant related only to the total number of the PHA channels. This number and the maximum PHA channel number are downloaded from the host and saved at X:QsortCyc and X:ChannelNum respectively. The height of a peak should passed on to the *PHA* process via register X1 and the PHA channel number is returned via register A. Address register r3 is used to index the PHA checking table. If a negative value in X1 is inputted, a zero will be returned. The source code of the PHA process is listed in Appendix C starting at 'PHA' on page 107.

The algorithm and implementation of this process reflected the concerns and

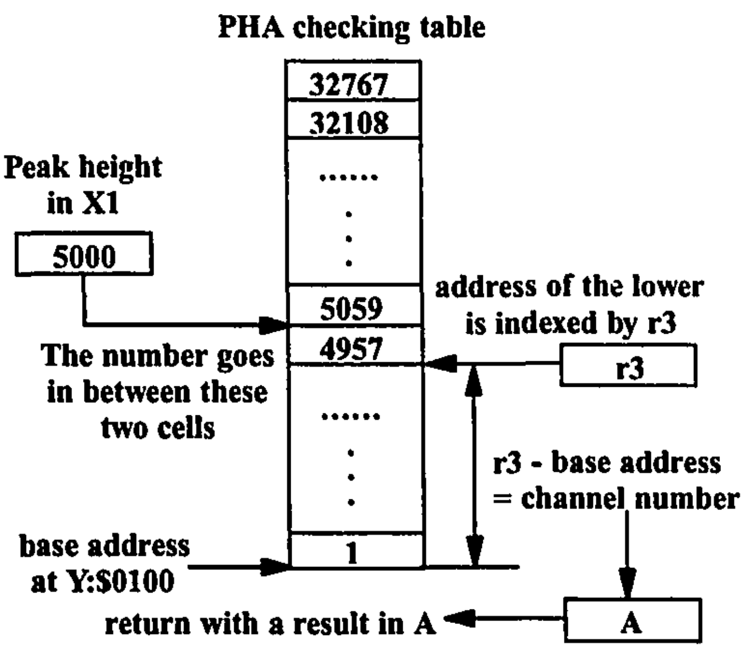


Figure 4.11 Schematic Diagram of the PHA process

emphases on the speed and efficiency of the process at the expense of some data memory space for the checking table. However, for real-time processing, this is an acceptable trade-off.

4.3.8. Real-time Data Transfer Process

The *real-time data transfer* process establishes an on-line data link between the host and the DSP-56. It was implemented as two general DSP to host data transfer utilities, which are referred to later in this section as *Transfer A* and *Transfer B*.

In general, *Transfer A* and *Transfer B* transmit a block of contiguous X or Y data memory to the host respectively. The starting address and the length of the memory block to be transferred are passed through the address register r4 and its offset register n4.

Both utilities are used by the control executive (**Figure 4.3**) to transfer 'peak buffers' to the host computer. In our application, the 'peak buffer A' is located in the X memory and the 'peak buffer B' in the Y memory (**Figure 4.10**). Therefore, *Transfer A* should be invoked when 'peak buffer A' is full and *Transfer B* when 'peak buffer B' is full. The decision which utility should be called is made by the control executive according to the status of the peak buffer full flags in the BuffStat register (**Figure 4.4**).

To fulfill the real-time data transfer, a parallel host process must be developed. Proper handshaking between the host and the DSP processes (**Figure 4.12**) is vital for real-time processing. In order not to delay the DSP process, the host processor must respond the DSP transfer request immediately, and thus an interrupt-driven process on

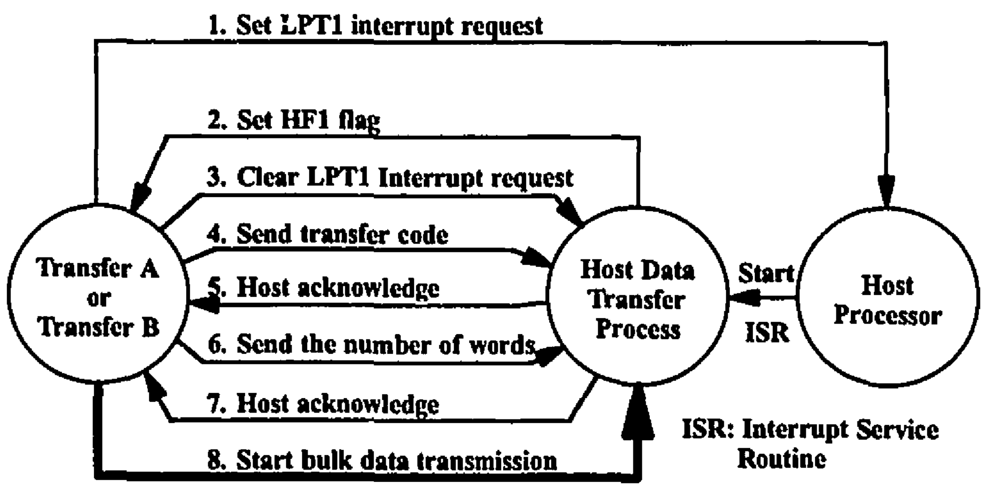


Figure 4.12 Real-time Data Transfer Between the Host and DSP

the host side is required. However there is no interrupt from the DSP-56 to the host processor in the implementation of the DSP-56 co-processor board currently in use. To overcome this, we used additional line to connect the DSP's auxiliary port to the PC's LPT1 parallel port. This connection allowed a TTL output from the auxiliary port to trigger the LPT1 interrupt on the PC. A host ISR was implemented to communicate with DSP.

As mentioned in **Section 3.1.5**, bit 19 of the Mode Control Register (MCR) controls the output of the TTL output. However bit manipulation instructions cannot be used here to control this bit. Data move instructions are applied to update the content of the MCR and thus control the TTL bit. Two words at X:TTL_Set and X:TTL_Clear, which have different status of the TTL control bit and keep the copies of the other bits of the MCR, are used as the sources of the MCR. Copying X:TTL_Set to MCR sets the TTL output high, sending the data transfer interrupt request to the host, and copying X:TTL_Clear to the MCR sets the TTL output low, clearing the interrupt request.

When either ***Transfer A*** or ***Transfer B*** is called, it first sends the data transfer interrupt request via the LPT1 interrupt. This immediately interrupts the host processor and activates the host data transfer process, which acknowledges the request by setting the host flag HF1 to the DSP processor. Then the DSP process clears the interrupt request and sends the host a transfer code, which tells the host the channel of the data and the transfer direction, 1 for channel A DSP to host data transfer, 2 for channel B DSP to host data transfer and 3 for host to DSP data transfer. After receiving the host acknowledgment, the DSP process sends the total number of 16-bit words to be transferred. And finally, after the host acknowledges for readiness, the DSP process starts bulk data transmission using the polling method (**Section 3.1.3.1**).

After the data transfer process successfully terminates, the control executive clears the corresponding peak BUFFER FULL flag and resets the write pointer of the 'peak buffer' and thus makes it empty and ready for further processing.

Because the polling technique is used in the bulk data transmission, the speed of the data transfer process on the host side is the governing factor of the overall data transfer rate. Special considerations have been taken into account on the host side, as discussed in the next section.

4.4. Host-DSP Interface for Real-time Data Transfer

As mentioned at the beginning of this chapter, the LiMCA software consists of three parts, viz. the DSP software, the host-DSP interface and the Graphical User Interface (GUI). Among them, the host-DSP interface directly communicates to the DSP processor and the GUI, providing a real-time data link between them. Its performance has a direct impact on the DSP process as well as the GUI. Speed and efficiency are the major concerns in the design and implementation of the interface.

4.4.1. General Views

The data processing on the host side includes (1) receiving the peak data from the DSP, (2) decoding the data, which are still in the DSP format, into proper C data type, for further host processing, (3) classifying the type of the peak using the decoded data, (4) analyzing the data statistically, (5) displaying the results graphically and interactively, and (5) saving the results for later references. These general tasks are decomposed into small functions. Most of them are foreground functions for most of the data processing tasks. Others are background functions controlled by the DSP interrupt requests and a few foreground functions used to install, enable and disable the interrupt-driven background functions. The background functions and the associated foreground functions were grouped together as the host-DSP interface. The rest formed the GUI and were programmed as foreground functions. As the speed of the host-DSP interface was the dominant concern, the number of tasks allocated to the interface was minimized. As a result, only task 1 was implemented in the interface, and the rest were left to the GUI. The host-DSP interface and the GUI are related implicitly. The main data communication between them is established through common memory blocks, managed by the GUI.

The interrupt related foreground functions of the interface are discussed in the next section. The background functions of the interface, dealing with the real-time data transfer, are described in **Section 4.4.3.** The common memory management between the interface and the GUI is detailed in **Section 4.4.4.**

4.4.2. Interrupt Installation and Control

As discussed in **Section 4.3.8**, an additional cable links the DSP's auxiliary port to the parallel port of the host PC, providing a physical interrupt source from the DSP board to the host. The cable and connectors are schematically shown in **Figure 4.13.** The connection as described in this figure connects the TTL output of the DSP

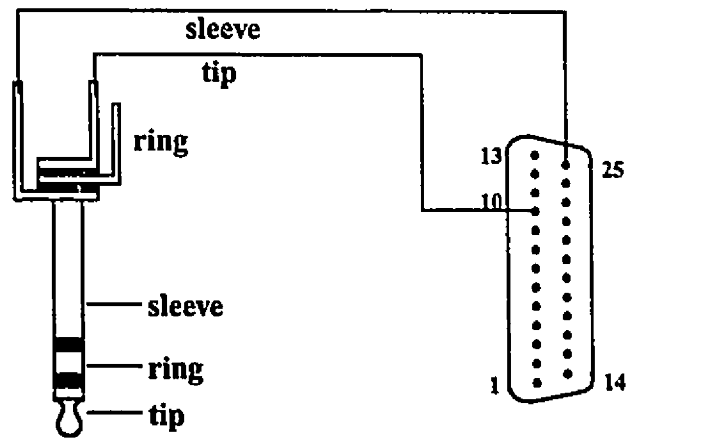


Figure 4.13 Cable Connection between DSP's Auxiliary Port and PC's Parallel Port

auxiliary port to the -ACK pin (pin 10) of the LPT1. This pin is routed to the hardware interrupt request 7 of the PC's programmable interrupt controller (PIC).

Two functions (`startInts()` and `stopInts()`) have been implemented to install a new interrupt handler, enable the interrupt, disable the interrupt and restore the old interrupt handler. The first two tasks were integrated in function `startInts()`. The other two were left to the function `stopInts(void)`. These functions replace an old interrupt handler with a new one and program the programmable interrupt controller (8259A PIC) and the first parallel printer interface (LPT1).

Installing the handler of the data transfer ISR is the first task of `startInts`. The data transfer process, which is invoked at the time of interrupt, was implemented as a group of functions with a tree type hierarchical structure. At the top of the tree was a function defined as interrupt type. A interrupt type pointer to this function was also defined to provide the entry address. This pointer is installed at a certain memory location in the PC's interrupt vector memory space. The locations where the pointer goes depend on the type of the interrupt. All the interrupt sources in PC have been enumerated by their interrupt numbers. In our case, hardware interrupt request line 7 is used. This interrupt has been assigned as interrupt 15. Consequently, the pointer to the function that services this interrupt must be installed at 003CH through 003FH, as each interrupt vector takes four memory slots starting from 0000H. In actual programming, placing the ISR handler is carried out by library functions provided by the compiler we are using, provided that the interrupt number is specified correctly. Note that the original handler must be saved before it is replaced with the new one.

The second task of the function is to enable the interrupt. This is done by enabling the LPT1 -ACK pin (pin 10) and enabling IRQ7 of the 8259A PIC. The port addresses of LPT1 start at 0378H through 037FH. The printer control register is located at 037AH, which is used to control the status of LPT1. Setting bit 4 of this register turns on the -ACK pin, thus the interrupt request can get through and reach the 8259A PIC. Other bits of the control register are irrelevant to our application and are ignored.

Before the interrupt can reach the PC's CPU, it must go through the 8259A PIC. Programming this controller for our application involves the manipulation of two 8-bit port registers at 0020H and 0021H. The second one is interrupt mask register, whose *n*th bit masks the interrupt request from line IRQ*n*. To enable IRQ7, which is used in our application, bit 7 should be cleared. Once an interrupt happens further interrupts are disabled automatically until the controller receives EOI (end of interrupt) code written to the first register at 0020H. This code itself is 0020H. However this must be sent by the background ISR, each time when it exits rather than this foreground function, to enable the following interrupts.

In summary, this function

- saves the old LPT1 ISR handler;
- installs the new LPT1 ISR handler, which services the host-DSP data transfer;
- enables the -ACK pin (pin 10) of LPT1 by setting bit 4 at 037AH;
- enables IRQ7 of the 8259A PIC by clearing bit 7 at 0021H.

The function `stopInts` does the opposite tasks as `startInts`. Briefly, it

- disables IRQ7 of the 8259A PIC by setting bit 7 at 0021H;
- disables the -ACK pin of LPT1 by clearing bit 4 at 037AH;
- discards the current LPT1 ISR handler and restores the old handler saved by `StartInts`;
- sends EOI to 0020H to make the 8259A PIC available for other interrupts.

4.4.3. Interrupt Service Routine (ISR) for Real-time Data Transfer

As mentioned in the previous section, the handler of the ISR is installed and enabled by the utility `startInts`. The ISR is activated when the DSP data transfer request occur through the LPT1 interrupt, and the foreground functions of the GUI are suspended until it completes the data transfer requested by the DSP process.

The ISR mainly deals with the host-DSP handshaking and data transfer. The handshaking sequence has already been discussed in **Section 4.3.8** and is shown in **Figure 4.12**. The data transfer process on the host side is undertaken in the polling mode. Before reading the port, it checks the RXDF bit of the Interrupt Status Register of the host port (**Figure 3.8**). If the flag is set, the routine reads the RXM and RXL in sequence and ignores the RXH, since the data transferred here are only 16-bit wide and take only two data ports. For details about data transfer between the host and the DSP in the polling mode, see **Section 3.1.3.1**. The data read from the RXM and RXL ports are being saved in the EMS memory space in the original DSP format for the GUI to process further. For the RXM and RXL ports, see **Section 3.1.3**, and the DSP data format, see **Figure 4.10**.

4.4.4. EMS (Expanded Memory Specification) Memory Pools for Real-time Peak Parameters

The real-time peak parameters from the host port are saved in two memory pools in the same format as in the DSP memory buffers. Each pool is for one channel (**Figure 4.14**). Note that only one of the pools is shown in this figure. These pools are accessible to the GUI. These memory pools are created in the EMS. The decision to set up the pools in the EMS was made based on the following considerations:

- The peak parameters have to be written to a storage media as fast as possible in order to catch up the fast DSP process. Thus RAM spaces were chosen for this purpose;
- Most of the PC's conventional memory space is occupied by system and application programs and data, and there is little room for massive data storage;
- The EMS is not being used in real-time data acquisition mode according to the GUI design of the LiMCA software, and it is much bigger than the conventional memory.

In each data acquisition, all the peak parameters are saved in the EMS pools. They are decoded for further analysis in real-time, and the results are displayed graphically and interactively. However the decoded data are not saved in real-time because the time constraints do not allow the access to a hard drive in real-time. The original data in the EMS pools are re-decoded and saved in a hard drive after the real-time data acquisition is completed. In this design, the size of the EMS pools are required to be big enough to accommodate all the peak parameters throughout from a whole acquisition.

Compared to the conventional memory, additional procedures are needed for the access of the EMS because of its structure. The whole EMS is divided into frames, which are further divided into pages. Each frame contains four consecutive pages, and each page has a memory space of 16 Kbytes. The PC's CPU can only access one EMS frame at a time. The frame that is currently addressed by the CPU is called the 'active frame'.

To access the EMS, the following generic procedures should be programmed in an application. They are:

(1) to get the total EMS pages and available pages of the current system;
(2) to get the EMS frame segment;
(3) to allocate the number of EMS pages needed to an EMS handler;
(4) to initialize a far pointer to the EMS frame segment;
(5) to map 4 consecutive EMS pages into the active frame;
(6) to access the active frame by pointers which are initialize by referencing the frame segment pointer set up in step (4);
(7) to map another 4 consecutive EMS pages into the active frame if the EMS pages in the current active frame is full, and to repeat step (6);
(8) to release the EMS before program exits.

These tasks have been implemented into utility functions in our application using MS-DOS interrupt 67H.

The EMS pool for channel A is shown in **Figure 4.14**. The EMS pool for channel B has the same structure. As one can see, 152 EMS pages, 2,490,368 bytes in

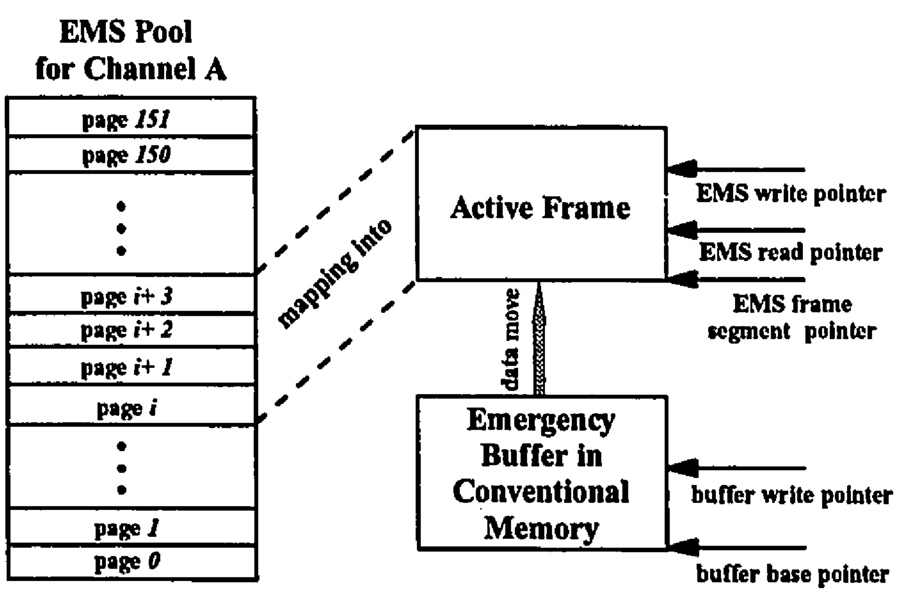


Figure 4.14 EMS Pool for DSP Real-time Peak Parameters of Channel A

total, are allocated to the EMS pool for channel A peak parameters. Noting that 16 bytes are used to describe a peak (**Figure 4.10**), each pool can save peak descriptions of up to 155,648 peak. Considering that it only takes several minutes to fill the sensing tube for the aluminum application, the size of the memory pools are more than enough for a data acquisition in this time range.

During the real-time data acquisition, the real-time data are written to the active frame by the background data transfer ISR via the EMS write pointer. The GUI reads the peak data from the active frame via the EMS read pointer. When the frame has been filled up by the data transfer ISR and has not yet been fully processed by the GUI, the data transfer ISR switches to a 16-Kbyte emergency buffer in conventional memory, so as not to stop the real-time data transfer process. After the GUI has processed the active frame, it maps another 4 pages into the active frame, moves the data from the emergency buffer, if there are any in the buffer, to the new pages in the active frame, and resets the EMS write and read pointers accordingly. In this way, the maximum delay of 1024 peaks is allowed between the DSP process and the host process. This time constraint should be considered in the implementation of the GUI.

As one can conclude that it is important that on the host side, the real-time data transfer process is not delayed in any circumstances, in order not to delay the DSP process. Between the real-time data transfer process and the data processing involved in the GUI, a buffer of adequate size in addition to the main storage media (the EMS pools in our case), for the real-time data is equally essential to allow some delay of the host process. Such time freedom is necessary for the complex data processing tasks assigned to the GUI.

4.5. Software Performance

Figure 4.15 shows the degree of utilization of the DSP co-processor board. The data were obtained by counting the total number of instructions along the longest branch in the final program. The calculation of the usage by all the processes in this Figure were based on the worst case data (see **Section 2.3.1** for the worst case operation). The DSP real-time software is assumed to be working in the stereo (two channel) mode. The ADC sampling rate is set to 50 KHz, which is adequate to avoid aliasing of the input analog signal. Based on the worst case operating conditions, i.e. 2000 peaks per second, the DSP processor is busy 49% of the total time.

In this calculation, two factors were not taken into account. The first is the length of the FIR (Finite Impulse Response) filter in the filter process (**Figure 4.2**) and the second is the number of cycles that are required to synchronize the DSP-host data transfer process. An increase in the length of the filter dramatically increases the time required by the filter process. In some cases, a sharp notch filter is needed to eliminate a narrow range of frequencies. Such a filter cannot be implemented in this software, because of the big number of taps required. A piece of high speed FIR filter hardware may be needed. With respect to the synchronization cycles, the data shown in **Figure 4.15** were calculated for a host computer with a 50 MHz system clock and a 100 nanosecond bus cycle. In this case, 3 waiting cycles are needed at the DSP level for each data transfer.

Up till now only about 50% of the DSP computational capacity is used. This

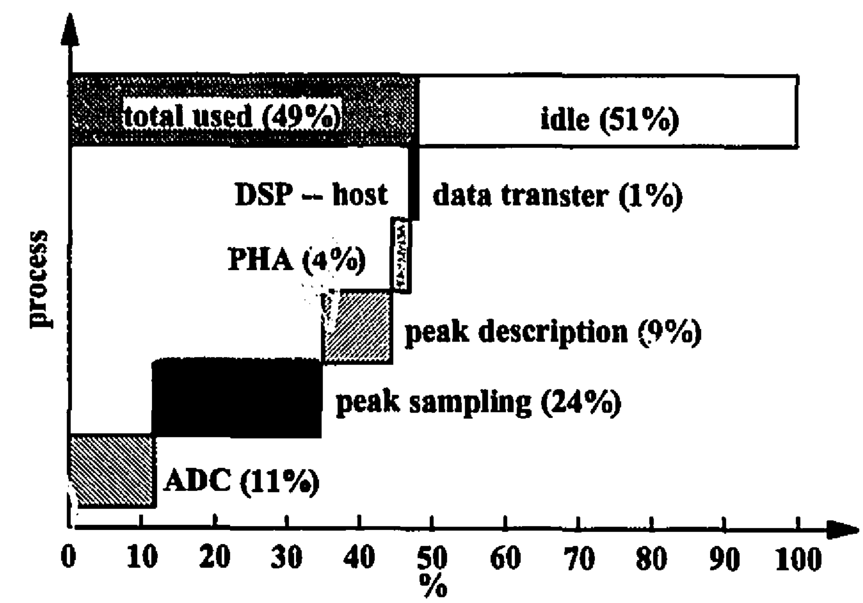


Figure 4.15 Usage of the DSP CPU

Table 4.3 Characteristics of LiMCA Peaks

peak type	start slope		end slope		width/height
	sign	value	sign	value	
NP	+	high	-	high	small
BJ	+	high	-	low	large
BF	+	low	-	uncertain	large
NBJ	-	high	+	low	large
NBF	-	low	+	uncertain	large
US	-	high	+	low	large

NP--**Normal Pulse** **NBJ** --**Negative Baseline Jump**
BJ --**Baseline Jump** **NBF** --**Negative Baseline Fluctuation**
BF--**Baseline Fluctuation** **US** --**Undershoot**

gives us the potential for future development, such as implementing the peak classification task at the DSP level and developing code to use the DAC channels for process control.

As for the host-DSP interface, many test runs for both water and molten aluminum showed that there were no detrimental delays introduced down to the DSP process from it. For the amount of the 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 foreground functions.

The peak description parameters, obtained by the DSP process and transferred to the host, can be used to characterize the different types of LiMCA peaks using **Table 4.3**. From this table, one can see that a simple peak classification algorithm can be used. It involves checking the sign of the peak and determining the relative magnitudes of the slopes at peak start and at peak end, and the peak width to height ratio. Successful classification depends upon using proper thresholds, which are currently determined experimentally.

In conclusion, in the implementation of the real-time software of our multi-processor system for LiMCA application, timing and communication are crucial factors. These concerns have been reflected in every phase of the software design and development. Proper measures used to tackle these concerns led to the successful completion of the real-time software including the DSP software and the host-DSP interface.

5. CONCLUSIONS AND FUTURE DEVELOPMENTS

5.1. Conclusions to the Thesis

- A DSP-based LiMCA system has been implemented to replace the first generation LiMCA system, which is based on the analog signal processing.
- The DSP real-time software and the host-DSP interface have been implemented and tested. They are sufficient to carry on the real-time LiMCA operation in the worst case.
- Enough computing capability of the DSP hardware and software are reserved for the future development, e.g. the implementation of the peak classification process at the DSP level.
- The EMS memory management has been implemented in the host-DSP interface. The use of the EMS in the communication with the DSP is crucial for the host computer to catch up the speed of the DSP process.
- A group of time domain peak description parameters are found to be useful and efficient for peak classification. A time domain real-time algorithm has been implemented in the DSP software to extract these parameters.
- A simple table-driven peak classification algorithm can be implemented according to the characteristics of the peaks described by the peak description parameters.

5.2. Suggestions for Future Work

To further enhance the performance the DSP LiMCA system, the following improvements are projected.

- A fast low-price DSP board is needed for the implementation of a sharp notch filter. Such filter is needed to filter out known frequency components that interfere with the LiMCA signal, in an industrial environment filled with electric noises from highly powered electric equipment. This filter could communicate with the DSP-56 board via its network port.
- For research purposes, it is required that the LiMCA peaks be sampled and saved along with their peak description parameters. However, the host-DSP interface can only handle the peak description parameters. The sampled peak must be transferred through other interface and be saved into the media control by the interface. A DSP

process can be implemented for the DSP-56 hardware to use its SCSI to save the peaks into a fast hard drive.

- Considering the number of peaks to be transferred and saved, a good compression algorithm and its implementation should be considered.
- Further studies on the peak classification algorithm must be conducted, especially on the classification of the Multiple Pulses.
- The classification algorithm should finally be implemented at the DSP level.
- To study the high pass filter effect and to compensate the magnitude attenuation of the LiMCA peaks, a software LiMCA signal simulator is needed.

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. Hutter, "An Evaluation of Aluminum 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.

[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.

[Carayannis and Shi 93] G. Carayannis and X. Shi, "Evaluating Metal Cleanliness Using DSP Technology", *Proc. of The International Conference on Signal Processing Applications & Technology*, ICSPAT'93, Santa Clara (California), pp. 895-904, Sept. 1993.

[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.

[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 and 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.

[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 Aluminum 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

[Nakajima 86] H. Nakajima, "On the Detection and Behaviour of Second Phase Particles in Steel Melts", *Ph.D Thesis*, Dept. of Mining and Metallurgical Eng., McGill University, 1986.

[Oppenheim and Schafer, 89] A.V. Oppenheim, R.W. Schafer, *Discrete-time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, pp118, 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.

[Siemensen 81] C.J. Siemensen, "Sedimentation Analysis of Inclusions in Aluminum 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.

[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-line Particle Detection System", *Master's Thesis*, Dept. of Mining & Metallurgical Eng., McGill University, 1992.

APPENDIX A: SPECIFICATIONS OF THE DSP-56 CO-PROCESSOR BOARD

CPU Type	Motorola DSP56001 Processor
system clock frequency	27 MHz
minimum instruction cycle	74 nano seconds
CPU architecture	parallel architecture, separate logic units, two Data Arithmetic Logic Units (ALU) for data manipulation, two Address Generation Units (AGU) for address generation and one program controller, multiple data and address buses, partition of data memory
data bus dynamic range	24 bit word width, 144 dB dynamic range
bus architecture	seven internal separate data and address buses supporting parallel data/address movement during execution of ALU/multiplier instructions
accumulator dynamic range	2 accumulators with 56 bit word width, 336 dB dynamic range
addressing	8 addressing pointers; Programmable auto-indexing supported with 8 offset registers; Modulo and reverse-carry addressing supported with 8 modulo registers
instructions	62 basic instructions; no-overhead DO-loops and repeated instructions are directly supported in the hardware.
memory	up to 64 Kwords (x24 bits) of storage for each of the X, Y and Program memory spaces
PC interface	The DSP-56 occupies seven 8-bit I/O ports whose base address is mapped by a header that accepts shorting plugs. It supports DMA using the DSP56001's built-in DMA facilities.
SCSI interface	up to 2 Mbytes/sec of 8-bit parallel I/O to external mass storage devices
DSP net interface	up to 2 Mbytes/sec of 24-bit parallel I/O to other DSP cards

Analog I/O	two channels of 16-bit analog to digital conversion, input sensitivity adjustable from 100 mV RMS to 776 mV RMS (280 mV to 2 volts peak-to-peak), sampling frequency software-controlled, 16 selections from 2 KHz to 100 KHz. A single channel, 12-bit, 400 KHz sample rate mode is also provided. Two channels of simultaneously sampled 16-bit digital-to-analog conversion with fixed (f_c = 20 KHz) 9^{th}-order elliptic reconstruction filters and sin(x)/x compensation are provided.
Auxiliary I/O	A three-conductor mini-phone jack mounted on the rear panel provides one-bit TTL level I/O interface to the DSP56001 chip.

APPENDIX B: THE PROTOTYPES OF THE DSP-56 INTERFACE FUNCTIONS

`extern void far configure_port_addresses(int baseAddr);`

Set port addresses of the DSP-56 processor.

`baseAddr`: input variable, the base port address of the processor.

`extern void far degmonParams(unsigned short *monStart, unsigned short *firstFree);`

Get parameters of the DEGMON monitor.

`monStart`: output variable, start address of the monitor;

`firstFree`: first address available after the monitor.

`extern int far do_host_command(int hc_addr);`

Execute host command.

`hc_addr`: input variable, the start address of the host command in DSP.

`extern void far DSP_Status( void );`

Get the status of the LiMCA process.

`extern void far empty_hp(void);`

Clear the host port.

`extern int far execute_instr(unsigned short startAddr);`

Start a DSP process through DEGMON monitor.

`startAddr`: input variable, start address of the DSP process.

`extern int far get_hp(unsigned long *data);`

Get a long data from the host port.

`data`: output variable, the data received.

`extern void far get_hpl(unsigned long *data);`

Get a long data from the host port without time-out.

`data`: output variable, the data received.

`extern void far get_hps(unsigned long *data);`

Get an int data from the host port.

`data`: output variable, the data received.

`extern void far get_port_addresses(unsigned *icr,unsigned *cvr,unsigned *isr, unsigned *hi, unsigned *mid, unsigned *lo);`

Get the port addresses of the DSP-56 processor `icr`: output variable, the address of the Interrupt Control Register;

`cvr`: output variable, the address of the Command Vector Register;

`isr`: output variable, the address of the Interrupt Status Register;

`hi`: output variable, the address of the Receive/Transmit Register (high byte);

`mid`: output variable, the address of the Receive/Transmit Register (middle byte);

`lo`: output variable, the address of the Receive/Transmit Register (low byte).

`extern void far hf0_off(void );`

Clear Host Flag 0.

`extern void far hf0_on(void );`

Set Host Flag 0.

`extern void far hf0_state(int *ret);`

Get the status of Host Flag 0.

`ret`: output variable, the status of Host Flag 0.

`extern void far hf1_off(void );`

Clear Host Flag 1.

`extern void far hf1_on(void );`

Set Host Flag 1.

`extern void far hf1_state(int *ret);`

Get Host Flag 1 status.

`ret`: output variable, the status of Host Flag 1.

`extern void far HostStop(void);`

Signal the DSP to stop the LiMCA process.

`extern int far if_hf2(void );`

Get the status of Host Flag 2.

`extern int far if_hf3(void );`

Get the status of Host Flag 3.

`extern int far LoadFile(char *fname, char **result, unsigned int *words, unsigned int *startAddr, int use_mon, int PMemEnable);`

Load a DSP program down to DSP-56 processor.

`fname`: input variable, file name of the compiled DSP process;

`result`: output variable, error message;

`words`: output variable, lengths of the DSP program;

`startAddr`: output variable, start address of the DSP program;

`use_mon`: input variable, YES if it is a boot load program, NO if not;

`PMemEnable`: input variable, YES to enable program memory, choose YES if it is a boot load program, NO if not.

```
extern void far read_hp( int channel );
```

Read peak parameters from the host port.

`channel`: output variable, channel of the DSP process.

Note: the peak parameters are saved in EMS.

```
extern int far read_memory(int space, unsigned short address, unsigned long
    *data);
```

Read the content of DSP memory.

`space`: input variable, which memory to read from, choices are P_SPACE, X_SPACE orY_SPACE;

`address`: input variable, address of the memory;

`data`: output variable, content of the memory.

```
extern int far readp(unsigned int addr, unsigned long *where);
```

Read the content of the DSP program memory.

`addr`: input variable, address of the memory;

`where`: output variable, content of the memory.

```
extern int far readx(unsigned int addr, unsigned long *where);
```

Read the content of the DSP X data memory.

`addr`: input variable, address of the memory;

`where`: output variable, content of the memory.

```
extern int far ready(unsigned int addr, unsigned long *where);
```

Read the content of the DSP Y data memory.

`addr`: input variable, address of the memory;

`where`: output variable, content of the memory.

```
extern void far reset_board(int EnableMemAfterReset);
```

Reset and start booting of the DSP processor.

`EnableMemAfterReset`: input variable, must always be TRUE for DSP-56

```
extern int far send_hp(unsigned long data);
```

Send a long data to host port.

`data`: input variable, the data to be sent.

```
extern  int far send_hp16(int data);
```

Send an int data to the host port without sign extension.

`data`: input variable, the data to be sent.

`extern int far send_hps(int data);`

Send an int data to the host port, the upper 8 bits are 0 extended if data >= 0 or 1 extended if data <0.

`data`: input variable, the data to be sent.

`extern void far terminate(void );`

Stop booting the DSP processor.

`extern void far write_hp(void);`

Write bulk data to the host port (to be developed).

`extern int far write_memory(int space, unsigned short address, unsigned long data);`

Write to DSP memory from the host port.

`space`: input variable, which memory to write to, choices are P_SPACE, X_SPACE or Y_SPACE;

`address`: input variable, address of the memory;

`data`: input variable, value to be written.

`extern int far writep(unsigned int addr, unsigned long data);`

Write to DSP program memory from the host port.

`addr`: input variable, address of the memory;

`data`: input variable, value to be written.

`extern int far writex(unsigned int addr, unsigned long data);`

Write to DSP X data memory from the host port.

`addr`: input variable, address of the memory;

`data`: input variable, value to be written.

`extern int far writey(unsigned int addr, unsigned long data);`

Write to DSP Y data memory from the host port.

`addr`: input variable, address of the memory;

`data`: input variable, value to be written.

APPENDIX C: DSP SOURCE CODE LISTING OF THE DSP LiMCA

```
Motorola DSP56000 Macro Cross Assembler  Version 3.02  93-11-23  22:06:55
lmcdsp.asm
             ;FILE: LMCDSP.ASM
             COMMENT @
*************************************************************************
MMPC
Dept. of Mining and Metullurgica Eng.
McGill University
(C) Copyright 1993
*************************************************************************
DSP-56 Processor card
DSP driver for LIMCA real-time data processing
Version 3.00   April, 1993
*************************************************************************
(C) 1990 MMPC, McGill University
Assemble with Motorola assembler:
asm56000 -A -B lmcdsp.lod -L lmcdsp
      Two Circular Buffers of length BUFSIZ are used to store the data for
ADC, one is in X_mem for channel A, the other one is in Y_mem for channel B.
      After initialization, this program waits in a command loop, where it
monitors the host port for a command data word. No action is taken until one
of the following commands appears. All other values are ignored.
      0: Record:   recording process from SSI to HI
      1: OnlineMCA:      real-time MCA
      2: OfflineMCA:     off-line MCA
      3: ReportStatus:   report process status to host
      4: SampleRate:     get sampling rate from host
      5: UpLoadMem:      upload DSP memory to host
      6: ZeroMem:  zero X and Y data memory
USES OF MEMORY
      X memory:
      $0000 -- $00FF for program varibles,  size: 256 wor
      $6000 -- $65FF for channel A peak-parameter buffer, size: 1.5k words
      $7000 -- $7FFF for channel A sampled peak buffer, size: 4k words
      $8000 -- $FBFF for channel A circular buffer, size: 31k words
      Y memory:
      $0000 -- $00FF for program varibles,  size: 256 words
      $0100 -- $04FF for channel B PHA table,  size: 1024 words
      $6000 -- $65FF for channel B peak-parameter buffer,  size: 1.5k words
      $7000 -- $7FFF for channel B sampled peak buffer, size: 4k words
      $8000 -- $FBFF for channel B circular buffer, size: 31k words
      P memory:
      $0000 -- $FFFF for program memory, size: 64k words
      END OF COMMENT SECTION @

             LIMCA     ident   3,0     ;LIMCA DATA PRO. DRIVER DSP-56
```

```
      opt      mex,cex,fc,rc      ;useful when a listing is produced.
      include      'lmcioeq.asm'      ;include the file of IO port equates
;--------------------------- constants --------------------------------
007C00 FIFOSIZE      EQU    31744 ;circular buffer size ,from $8000 to $FBFF
007000 PK_SAMPLE_START      EQU    $7000 ;start addr. of pk sampled data buffer
000100 PHATABLESTART        EQU    $0100 ;start addr. of PHA scaling table
000080 PEAKDBFSIZE   EQU    $0080 ;peak-parameter buffer size
006000 PEAKDBFSTART         EQU    $6000 ;start addr. of peak-parameter buffer
000000 DSPAHITX      EQU    $0    ;DSP -> HI data tranfer code for chanA
000001 DSPBHITX      EQU    $1    ;DSP -> HI data tranfer code for chanB
000002 DSPHIRV              EQU    $2    ;HI -> DSP data tranfer code
;bit 0 for tranfer channel: 0 for chanA, 1 for chanB bit 1 for tranfer
;direction: 0 for DSP -> HI, 1 for HI -> DSP
;----------------bit symbols in ProcStatus Register--------------------
000000 HOSTSTOP      EQU    0      ;host PC stop flag
000001 CHANNELA      EQU    1      ;channel A flag
000002 CHANNELB      EQU    2      ;channel B flag
000003 TIMEUP EQU    3      ;time flag, 1: exceed the user-specified time
;----------------bit symbols in BuffStatus Register--------------------
000000 FIFOFULL      EQU    0      ;FIFO buffer full flag
000001 FIFOEMPTY     EQU    1      ;FIFO buffer empty flag
000004 PEAKDBFAFULL EQU     4      ;peak-parameter buffer A full flag
000005 PEAKDBFBFULL EQU     5      ;peak-parameter buffer B full flag
;----------------bit symbols in PkSamplingSt Register--------------------
000000 PK_SAMPLING_FIND_A         EQU      0
000001 PK_SAMPLING_SIGN_A         EQU      1
000002 PK_SAMPLING_FINISH_A       EQU      2
000008 PK_SAMPLING_FIND_B         EQU      8
000009 PK_SAMPLING_SIGN_B         EQU      9
00000A PK_SAMPLING_FINISH_B       EQU      10
;----------------bit symbols in PkSamplingCr Register--------------------
000000 PK_SAMPLING_CHANNEL_A      EQU    0
000001 PK_SAMPLING_CONT_A         EQU    1
000008 PK_SAMPLING_CHANNEL_B      EQU    8
000009 PK_SAMPLING_CONT_B         EQU    9
;------------------------variables-----------------------------------
X:0000        ORG      X:$0
 ;--------------- global varibles, status and control regs. ------------
d   X:0000 000000     ProcStatus    DC     0      ;process status
d   X:0001 000000     BuffStatus    DC     0      ;buffer status
d   X:0002 000000     PkSamplingSt DC      0      ;pk sampling status register
;-------------- parallel varibles, status and control regs. -----------
;for pk sampling process
d   X:0003 000000     PkSampleWriteA      DC     0       ;sampled peak buffer A
                                                          ;write pointer
d   X:0004 000000     PkSamplePreVaA      DC     0       ;previous value at the
                                                          ;point before peak start
d   X:0005 000000     PkStartLo16A DC     0      ;pk start low 16 bits
d   X:0006 000000     PkStartHi16A DC     0      ;pk start high 16 bits
d   X:0007 000000     PkWidthA      DC     0      ;pk width count for pk
                                                  ;description process
d   X:0008 000000     PkBufferWriteA      DC     0       ;peak buffer A write ptr
d   X:0009 000000     PkBuffCntA    DC     0      ;pk buffer A counter
```

```
;---------------------- other varibles -------------------------------
;for pk sampling process
d  X:000A 000000   NoiseHi            DC     0
d  X:000B 000000   NoiseLo            DC     0
;for PHA process
d  X:000C 000000   QsortCyc     DC     0      ;number of sorting cycles for
                                              ;PHA
d  X:000D 000000   ChannelNum   DC     0      ;total number of PHA channels
                   ;Variables of the program frame and the circular buffers
d  X:000E 000000   FUNCTION     DC     0      ;current function #
d  X:000F 900000   MODELATCH    DC     $900000
                   ;a copy of mode latch, 50 kHz is default sampling rate
   COMMENT *
      details of mode latch:
            bit 16 = DSPNET bus request
            bit 17 = serial output line
            bit 18 = srate select: 0 = normal, 1 = high speed
            bit 19 = interrupt mode: 0 = SCSI, 1 = DSPNET
            bits 20..23 = srate select*
d  X:0010 000000   TTL_Set      DC     0
d  X:0011 000000   TTL_Clear    DC     0
d  X:0012 000253   fList        DC     Record       ;fcn code 0
d  X:0013 000254          DC    OnlineMCA
d  X:0014 0002BC          DC    OfflineMCA
d  X:0015 0002E1          DC    ReportStatus
d  X:0016 0002F4          DC    SampleRate
d  X:0017 000300          DC    UpLoadMem
d  X:0018 00032A          DC    ZeroMem
d  X:0019 0002E0          DC    NULL
d  X:001A 008000   FIFORead     DC     $8000  ;circular buffer read pointer at
                                              ;the first addr
d  X:001B 000000   CountLo16    DC     0      ;lower 16 bits of the total data
                                              ;count
d  X:001C 000000   CountHi16    DC     0      ;upper 16 bits of the total data
                                              ;count
   ;These two time labels point to the data point just processed, not the the
   ;data point about to be processed.
d  X:001D 00FFFF   CountLo16Max DC     $FFFF  ;max. of total data count set by
                                              ;host (low)
d  X:001E 00FFFF   CountHi16Max DC     $FFFF  ;max. of total data count set by
                                              ;host (high)
Y:0000        ORG    Y:$0
;--------------- global varibles, status and control regs. ------------
d  Y:0000 000000   CommandWord  DC     0      ;host PC command word is saved
                                              ;here
d  Y:0001 000000   FIFOAdvance  DC     0      ;circular write pointer advance
                                              ;counter
d  Y:0002 000000   PkSamplingCr DC     0      ;pk sampling control register
;-------------- parallel varibles, status and control regs. -----------
;for pk sampling process
d  Y:0003 000000   PkSampleWriteB     DC     0      ;sampled peak buffer B
                                                    ;write pointer
d  Y:0004 000000   PkSamplePreVaB     DC     0      ;previous value at the
```

```
                                                          ;point before peak start
d  Y:0005 000000   PkStartLo16B DC     0      ;pk start low 16 bits
d  Y:0006 000000   PkStartHi16B DC     0      ;pk start high 16 bits
d  Y:0007 000000   PkWidthB     DC     0      ;pk width count
;for pk description process
d  Y:0008 000000   PkBufferWriteB     DC     0      ;peak buffer B write ptr
d  Y:0009 000000   PkBuffCntB   DC     0      ;pk buffer B counter
;---------------------- other varibles ------------------------------
;Stacks for SSI ISR
d  Y:000A 000000   Stack_a1           DC     0
d  Y:000B 000000   Stack_y0           DC     0

P:0090       ORG    P:$90
;Real-time code in low memory for best efficiency
                   COMMENT *
note: it's important that all this code (at least the a ctual real-time parts
of it) reside in low memory.     *
;--------------- Interrupt Service Routine (ISR) for ADC --------------
;The ADC interrupt routine copies the ADC data to the Circular Buffer.
;Register r7 is used as an advance counter for the delay between write pointer
;and read pointer. r0 is used as the Circular Buffer write pointer. They are
;not stacked so that They should not ;be used for other purposes.
SSIDataInPtr
P:0090  0D0091     jsr    <SSIDataIn
SSIDataIn
P:0091  0AAE83     jclr   #M_RFS, X:<<M_SR, SSID_chanB
        000095
P:0093  0860AF     movep  X:<<M_RX, X:(r0)    ;save data in X FIFO buffer
P:0094  000004     rti
SSID_chanB
P:0095  0858EF     movep  X:<<M_RX, Y:(r0)+   ;save data in Y FIFO buffer
P:0096  045F17     lua    (r7)+, r7    ;update write pointer advance counter
P:0097  4E0B00     move   y0, Y:Stack_y0      ;push y0 register
P:0098  5C0A00     move   a1, Y:Stack_a1      ;push a1 register
P:0099  46F400     move   #>FIFOSIZE, y0
        007C00
P:009B  22EC00     move   r7, a1
P:009C  4E8B53     eor    y0, a Y:Stack_y0, y0      ;pop y0
P:009D  0AF0A2     jne    SSID_Ret
        0000A2            ;if FIFO is not overflow, return, otherwise
P:009F  0A0120     bset   #FIFOFULL, X:<BuffStatus  ;set FIFO full flag.
P:00A0  0BF080     jsr    HostStop
        00035E                                      ;stop the process
SSID_Ret
P:00A2  5C8A00     move   Y:Stack_a1, a1            ;pop a1
P:00A3  000004     rti                        ;interrupt process complete
;-------------------------------------------------------------------------
HostStopPtr
P:00A4  0D00A5     jsr    <HostStopInts
HostStopInts
P:00A5  0BF080     jsr    StopInts
        00034E
P:00A7  0A0020     bset   #HOSTSTOP, X:<ProcStatus
```

```
P:00A8  000004      rti
;----------------------REAL TIME SUBROUTINES---------------------------
PkSampling
P:00A9  478A00      move  X:NoiseHi, y1
P:00AA  468B00      move  X:NoiseLo, y0
P:00AB  45F400      move  #>$8000, x1
        008000            ;factor for shifting data right 8 bits
P:00AD  05F421      move  #>FIFOSIZE-1, m1
        007BFF            ;make r1 modulo of 31k
P:00AF  619A00      move  X:FIFORead, r1
                           ;r1 is circular buffer read pointer for channel A
P:00B0  63F400      move  #>1, r3
        000001            ;r3 is channel A pk width counter
P:00B2  227513      clr   a      r3, r5 ;r5 is channel B pk width counter
P:00B3  540200      move  a1, X:PkSamplingSt ;reset peak sampling status reg.
P:00B4  76F400      move  #>2, n6       ;in rare cases, two peaks are close
        000002            ;together, r6 and n6 are use to help retrieve
                          ;previous value for the following peaks
_ChannelAInit
P:00B6  0A02C0      jclr  #PK_SAMPLING_CHANNEL_A,Y:PkSamplingCr,_ChannelBInit
        0000C0            ;if not channelA, go check channel B
P:00B8  62F400      move  #>PK_SAMPLE_START, r2
        007000            ;r2 is peak buffer A write pointer
P:00BA  0A02C1      jclr  #PK_SAMPLING_CONT_A, Y:PkSamplingCr, _ChannelBInit
        0000C0            ;If  set: continue old pk, clr: start a new pk
P:00BC  0A0220      bset  #PK_SAMPLING_FIND_A, X:PkSamplingSt
                    ;a pk is found, doesn't necessarily mean a pk is finished
P:00BD  0A0241      bclr  #PK_SAMPLING_CONT_A, Y:PkSamplingCr
P:00BE  628300      move  X:PkSampleWriteA, r2       ;resume the pk sampling A
                                                     ;write pointer
P:00BF  638700      move  X:PkWidthA, r3      ;load the width needed to be
                                              ;continued
_ChannelBInit
P:00C0  0A02C8      jclr  #PK_SAMPLING_CHANNEL_B, Y:PkSamplingCr, _Loop
        0000CA
P:00C2  64F400      move  #>PK_SAMPLE_START, r4
        007000            ;r4 is peak buffer B write pointer
P:00C4  0A02C9      jclr  #PK_SAMPLING_CONT_B, Y:PkSamplingCr, _Loop
        0000CA      ;If it is set: continue old pk, clr: start a new peak
P:00C6  0A0228      bset  #PK_SAMPLING_FIND_B, X:PkSamplingSt
                    ;a pk is found, doesn't necessarily mean a pk is finished
P:00C7  0A0249      bclr  #PK_SAMPLING_CONT_B, Y:PkSamplingCr
P:00C8  6C8300      move  Y:PkSampleWriteB, r4       ;resume the pk sampling B
                                                     ;write pointer
P:00C9  6D8700      move  Y:PkWidthB, r5      ;load the width needed to be
                                              ;continued
_Loop
P:00CA  22EE00      move  r7, a
P:00CB  57F400      move  #>1, b
        000001
P:00CD  205705      cmp   b, a   (r7)-  ;test if the circular buffer is empty,
P:00CE  0AF0A7      jgt   _ChannelA    ;if not empty, continue
        0000D8            ;if yes, check if host stopped the process
```

```
P:00D0  045F17      lua     (r7)+, r7      ;if not stopped by host, go back and
                                           ;continue
P:00D1  0A0080      jclr    #HOSTSTOP, X:<ProcStatus, _Loop
        0000CA
P:00D3  0A0121      bset    #FIFOEMPTY, X:<BuffStatus ;do cleanups, and return
P:00D4  611A00      move    r1, X:FIFORead      ;save circular buffer read
                                                ;pointer
P:00D5  05F421      move    #-1, m1
        FFFFFF                                  ;reset to linear addressing
P:00D7  00000C      rts
_ChannelA
P:00D8  0A02C0      jclr    #PK_SAMPLING_CHANNEL_A, Y:PkSamplingCr, _ChannelB
        000118                         ;if not channel A, go check channel B
P:00DA  44E113      clr     a      X:(r1), x0
        ;fetch a data from the circular buffer, don't update theread pointer
P:00DB  2000A0      mpy     x1, x0, a    ;shift the data 8 bits right
P:00DC  0A02A0      jset    #PK_SAMPLING_FIND_A, X:PkSamplingSt, _ContPkA
        0000FB
_NewPkA                     ;to find a new peak, save the pre-value, start-value
                            ;and the 'start time
P:00DE  448475      cmp     y1, a  X:PkSamplePreVaA, x0      ;compare the data
                                                             ;with NoiseHi
P:00DF  0AF0AF      jle     _negA
        0000EB
P:00E1  445A00      move    x0, X:(r2)+  ;save the value at the point before
                                         ;peak start
P:00E2  545A00      move    a1,X:(r2)+   ;save the value at peak start
P:00E3  559B00      move    X:CountLo16, b1
P:00E4  550500      move    b1, X:PkStartLo16A ;save the peak start point low
                                               ;16 bits
P:00E5  559C00      move    X:CountHi16, b1
P:00E6  550600      move    b1, X:PkStartHi16A ;save the peak start point hight
                                               ;16 bit
P:00E7  0A0220      bset    #PK_SAMPLING_FIND_A, X:PkSamplingSt
                    ;a positive pk is found
P:00E8  0A0221      bset    #PK_SAMPLING_SIGN_A, X:PkSamplingSt
P:00E9  0AF080      jmp     _ChannelB
        000118
_negA
P:00EB  200055      cmp     y0, a  ;compare the data with NoiseLo
P:00EC  0AF0A1      jge     _finA
        0000F8
P:00EE  445A00      move    x0, X:(r2)+  ;save the value at the point before
                                         ;peak start
P:00EF  545A00      move    a1,X:(r2)+   ;save the value at peak start
P:00F0  559B00      move    X:CountLo16, b1
P:00F1  550500      move    b1, X:PkStartLo16A ;save the peak start point low
                                               ;16 bit
P:00F2  559C00      move    X:CountHi16,b1
P:00F3  550600      move    b1,X:PkStartHi16A  ;save the peak start point hight
                                               ;16 bit
P:00F4  0A0220      bset    #PK_SAMPLING_FIND_A, X:PkSamplingSt
                    ;a negative pk is found
```

```
P:00F5  0A0201      bclr  #PK_SAMPLING_SIGN_A, X:PkSamplingSt
P:00F6  0AF080      jmp   _ChannelB
        000118
 _finA                    ;if neither a neg. nor a pos. pk is found
P:00F8  560400      move  a, X:PkSamplePreVaA        ;uppdate the pre-value
P:00F9  0AF080      jmp   _ChannelB
        000118
_ContPkA                  ;continue to find pk end and pk width
P:00FB  545A00      move  a1, X:X:(r2) ;save pk value
P:00FC  0A0281      jclr  #PK_SAMPLING_SIGN_A, X:PkSamplingSt, _neg2A
        000107
P:00FE  205B75      cmp   y1, a  (r3)+ ;compare with NoiseHi for pos. pk
P:00FF  0AF0A1      jge   _ChannelB
        000118            ;if greater than NoiseHi, it is not finished
P:0101  0A0222      bset  #PK_SAMPLING_FINISH_A, X:PkSamplingSt
                          ;set pk A finished flag
P:0102  560455      cmp   y0, a  a, X:PkSamplePreVaA
                          ;update pre-value for next peak
P:0103  0AF0A9      jlt   _foloA ;if the present point is smaller than NoiseLo
        000110                   ;it is followed immediately a negative peak
P:0105  0AF080      jmp   _ChannelB
        000118
_neg2A
P:0107  205B55      cmp   y0, a  (r3)+  ;compare with NoiseLo for neg. pk
P:0108  0AF0AF      jle   _ChannelB
        000118            ;if smaller than NoiseLo, it is not finished
P:010A  0A0222      bset  #PK_SAMPLING_FINISH_A, X:PkSamplingSt
                          ;set pk A finished flag
P:010B  560475      cmp   y1, a  a, X:PkSamplePreVaA
                          ;update pre-value for next peak
P:010C  0AF0A7      jgt   _foloA ;if the present point is bigger than NoiseHi
        000110                   ;it is followed immediately a positive peak
P:010E  0AF080      jmp   _ChannelB
        000118
_foloA
P:0110  225600      move  r2, r6 ;get a copy of sampled peak write pointer
P:0111  045F17      lua   (r7)+, r7
P:0112  044616      lua   (r6)-n6, r6  ;rewind this pointer to the second
        ;point to the last, the point will be the starting point of next peak
P:0113  045515      lua   (r5)-, r5
                          ;rewind PkWidthB point, since next time the present
                          ;point has to be reprocessed
P:0114  56E600      move  X:(r6), a
                          ;fetch the second to last data of present pk
P:0115  560400      move  a, X:PkSamplePreVaA
P:0116  0AF080      jmp   _exit
        00017A            ;exit directly
_ChannelB
P:0118  0A02C8      jclr  #PK_SAMPLING_CHANNEL_B, Y:PkSamplingCr,
        000158            _AdrUpdate
                          ;if not channel B, go to update address pointers
P:011A  4CE113      clr         a      Y:(r1), x0
        ;fetch a data from the circular buffer, don't update the read pointer
```

```
P:011B  2000A0     mpy    x1, x0, a    ;shift the data 8 bits right
P:011C  0A02A8     jset         #PK_SAMPLING_FIND_B, X:PkSamplingSt, _ContPkB
        00013B
_NewPkB                   ;to find a new peak, save the pre-value, start-value
                          ;and the start-time
P:011E  4C8475     cmp    y1, a  Y:PkSamplePreVaB, x0
                          ;compare the data with NoiseHi
P:011F  0AF0AF     jle    _negB
        00012B
P:0121  4C5C00     move   x0, Y:(r4)+
                          ;save the value at the point before pk start
P:0122  545C00     move   a1, X:(r4)+  ;save the value at peak start
P:0123  559B00     move   X:CountLo16, b1
P:0124  5D0500     move   b1, Y:PkStartLo16B
                          ;save the pk start point low 16 bit
P:0125  559C00     move   X:CountHi16, b1
P:0126  5D0600     move   b1, Y:PkStartHi16B
                          ;save the pk start point hight 16 bit
P:0127  0A0228     bset   #PK_SAMPLING_FIND_B, X:PkSamplingSt
                          ;a positive pk is found
P:0128  0A0229     bset    #PK_SAMPLING_SIGN_B, X:PkSamplingSt
P:0129  0AF080     jmp    _AdrUpdate
        000158
_negB
P:012B  200055     cmp    y0, a  ;compare the data with NoiseLo
P:012C  0AF0A1     jge    _finB
        000138
P:012E  445C00     move   x0, X:(r4)+
                          ;save the value at the point before pk start
P:012F  545C00     move   a1, X:(r4)+  ;save the value at peak start
P:0130  559B00     move   X:CountLo16, b1
P:0131  5D0500     move   b1, Y:PkStartLo16B
                          ;save the pk start point low 16 bits
P:0132  559C00     move   X:CountHi16, b1
P:0133  5D0600     move   b1, Y:PkStartHi16B
                                 ;save the pk start point hight 16 bits
P:0134  0A0228     bset   #PK_SAMPLING_FIND_B, X:PkSamplingSt
                          ;a negative pk is found
P:0135  0A0209     bclr   #PK_SAMPLING_SIGN_B, X:PkSamplingSt
P:0136  0AF080     jmp    _AdrUpdate
        000158
_finB
P:0138  5E0400     move   a, Y:PkSamplePreVaB
              ;if neither a neg. nor a pos. pk is found, uppdate the pre-value
P:0139  0AF080     jmp    _AdrUpdate
        000158
_ContPkB                  ;continue to find pk end and pk width
P:013B  5C5C00     move   a1, Y:(r4)+  ;save pk value
P:013C  0A0289     jclr   #PK_SAMPLING_SIGN_B, X:PkSamplingSt, _neg2B
        000147
P:013E  205D75     cmp    y1, a  (r5)+ ;compare with NoiseHi for pos. pk
P:013F  0AF0A1     jge           _AdrUpdate
        000158            ;if greater than NoiseHi, it is not finished
```

```
P:0141  0A022A   bset   #PK_SAMPLING_FINISH_B, X:PkSamplingSt
                        ;set pk A finished flag
P:0142  5E0455   cmp    y0, a  a, Y:PkSamplePreVaB
                        ;update pre-value for next peak
P:0143  0AF0A9   jlt    _foloB ;if the present point is smaller than NoiseLo
        000150          ;it is followed immediately a negative peak
P:0145  0AF080   jmp    _AdrUpdate
        000158
_neg2B
P:0147  205D55   cmp    y0, a  (r5)+  ;compare with NoiseLo for neg. pk
P:0148  0AF0AF   jle    _AdrUpdate
        000158          ;if smaller than NoiseLo, it is not finished
P:014A  0A022A   bset   #PK_SAMPLING_FINISH_B, X:PkSamplingSt
                        ;set pk A finished flag
P:014B  5E0475   cmp    y1, a  a, Y:PkSamplePreVaB
                        ;update pre-value for next peak
P:014C  0AF0A7   jgt    _foloB ;if the present point is bigger than NoiseHi
        000150                 ;it is followed immediately a positive peak
P:014E  0AF080   jmp    _AdrUpdate
        000158
_foloB
P:0150  229600   move   r4, r6 ;get a copy of sampled peak write pointer
P:0151  045F17   lua           (r7)+, r7
P:0152  044616   lua    (r6)-n6, r6  ;rewind this pointer to the second
        ;point to the last, the point will be the starting point of next peak
P:0153  045313   lua    (r3)-, r3           ;rewind PkWidthA one point,
                 ;since next time the present point has to be reprocessed
P:0154  5EE600   move   Y:(r6), a
                        ;fetch the second to last data of present pk
P:0155  5E0400   move   a, Y:PkSamplePreVaB
P:0156  0AF080   jmp    _exit
        00017A          ;exit directly
_AdrUpdate
P:0158  569C00   move   X:CountHi16, a
P:0159  449E00   move   X:CountHi16Max, x0
P:015A  579B45   cmp    x0, a  X:CountLo16, b
;compare the high 16 bit data count with the max high 16 bit data count
P:015B  0AF0A2   jne    _NotTimeUp
        000169
P:015D  449D00   move   X:CountLo16Max, x0
P:015E  20004D   cmp    x0, b
                        ;compare the low 16 bit data count with the max.
P:015F  0AF0A2   jne    _NotTimeUp
        000169
P:0161  0BF080   jsr    HostStop
        00035E          ;stop SSI interrupt
P:0163  0A0023   bset   #TIMEUP, X:ProcStatus     ;set the time up flag
P:0164  370000   move   #0, r7 ;maipulate r7 to make the circular buffer
                               ;empty and to discard the data after time up
P:0165  611A00   move   r1, X:FIFORead      ;save FIFO read pointer
P:0166  05F421   move   #-1, m1
        FFFFFF          ;resume linear addressing mode of r1
P:0168  00000C   rts
```

```
_NotTimeUp
P:0169  44F400      move   #>1, x0
        000001
P:016B  44F448      add    x0, b  #>$10000, x0
        010000             ;increment low 16 data count
P:016D  20594D      cmp    x0, b  (r1)+
                           ;see if it is overflow, and update buffer readptr
P:016E  0AF0A2      jne    _NoCarry
        000174
P:0170  44F400      move   #>1, x0
        000001
P:0172  2F0040      add    x0, a  #0, b ;high count plus 1, clr low count
P:0173  541C00      move   a1, X:CountHi16     ;save high 16 data count
_NoCarry
P:0174  551B00      move   b1, X:CountLo16     ;save low 16 data count
P:0175  0A02A2      jset   #PK_SAMPLING_FINISH_A, X:PkSamplingSt, _exit
        00017A                             ;check exit conditions
P:0177  0A02AA      jset   #PK_SAMPLING_FINISH_B, X:PkSamplingSt, _exit
        00017A
P:0179  0C00CA      jmp    _Loop         ;go back looping
_exit
P:017A  611A00      move   r1, X:FIFORead
                           ;save circular buffer read pointer
P:017B  05F421      move   #-1, m1
        FFFFFF             ;resume linear addressing mode
P:017D  620300      move   r2, X:PkSampleWriteA
P:017E  6C0300      move   r4, Y:PkSampleWriteB
                           ;save sampled pk write ptrs
P:017F  630700      move   r3, X:PkWidthA
P:0180  6D0700      move   r5, Y:PkWidthB             ;save pk widths
P:0181  0A02A2      jset   #PK_SAMPLING_FINISH_A, X:PkSamplingSt,
        000184             _SetContB
P:0183 0A0261       bset          #PK_SAMPLING_CONT_A, Y:PkSamplingCr
_SetContB
P:0184  0A02AA      jset   #PK_SAMPLING_FINISH_B, X:PkSamplingSt, _Ret
        000187
P:0186  0A0269      bset   #PK_SAMPLING_CONT_B, Y:PkSamplingCr
P:0187  00000C      _Ret   rts
;-------------------------------------------------------------------
PkDescription
P:0188  0A0282      jclr   #PK_SAMPLING_FINISH_A, X:PkSamplingSt,
        0001CC             _ChannelB
P:018A  628800      move   X:PkBufferWriteA, r2
                           ;r2 is pk buffer writer ptr
P:018B  63F400      move   #>PK_SAMPLE_START, r3
        007000             ;r3 is pk sampled buffer read pointer
P:018D  227100      move   r3, r1
             ;r1 keeps pointing the start address of the sampled peak buffer
P:018E  668900      move   X:PkBuffCntA, r6    ;r6 is pk buffer counter
P:018F  76F400      move   #>8, n6
        000008             ;n6 is the number of parameters per pk
;calucalate start slope
P:0191  44DB00      move   X:(r3)+,x0   ;first point to x0
```

```
P:0192  56E300      move   X:(r3),a
                           ;second point to a, note that r3 is not incremented
P:0193  468644      sub    x0, a X:PkStartHi16A, y0
P:0194  465A00      move   y0, X:(r2)+
             ;calculate the start slope and save the start point high 16 bits
P:0195  468500      move   X:PkStartLo16A, y0
P:0196  465A00      move   y0, X:(r2)+  ;save start point low 16 bit
P:0197  565A00      move   a, X:(r2)+
                           ;save start slope find peak max. time at max.
P:0198  64F400      move   #>1, r4
        000001             ;r4 here is a counter
P:019A  350000      move   #0, r5       ;r5 keeps the count at max.
P:019B  468700      move   X:PkWidthA, y0
P:019C  0A0281      jclr   #PK_SAMPLING_SIGN_A, X:PkSamplingSt, _NegMaxA
        0001AB
P:019E  45DB00      move   X:(r3)+, x1  ;x1 keeps the max. value
P:019F  56DB00      move   X:(r3)+, a   ;a keeps the present value
_PosMaxLoopA
P:01A0  199B65      cmp    x1, a X:(r3)+, a   a, y1
        ;compare the current to the max., keep the current in y1 and update A
P:01A1  0AF0AF      jle    _PUpdateA
        0001A5
P:01A3  20E500      move   y1, x1
                           ;if the new data is larger than the present max.
P:01A4  229500      move   r4, r5 ;update the max.
_PUpdateA
P:01A5  045C14      lua    (r4)+, r4    ;update r4 counter
P:01A6  228F00      move   r4, b
P:01A7  22995B      eor    y0, b r4, n1 ;check peak end, n1 keeps track of r4
;counter, which is used later to offset r3 to the end of sample peak buffer
P:01A8  0E21A0      jne    _PosMaxLoopA ;note that pk width is in y0 now
P:01A9  0AF080      jmp    _EndSlopeA
        0001B6
_NegMaxA
P:01AB  45DB00      move   X:(r3)+, x1  ;x1 keeps the max. value
P:01AC  56DB00      move   X:(r3)+, a   ;a keeps the present value
_NegMaxLoopA
P:01AD  199B65      cmp    x1, a X:(r3)+, a   a, y1  ;compare the current to
                           ;the max., keep the current in y1 and update a
P:01AE  0AF0A1      jge    _NUpdateA
        0001B2
P:01B0  20E500      move   y1, x1 ;if the new data is smaller than the present
P:01B1  229500      move   r4, r5 ;max., update the max. and data count at max
_NUpdateA
P:01B2  045C14      lua           (r4)+, r4    ;update r4 counter
P:01B3  228F00      move   r4, b
P:01B4  22995B      eor    y0, b  r4, n1 ;check end point, n1 keeps track of r4
        ;counter, which is used later to offset r3 to the end of sample peak
P:01B5  0E21AD      jne    _NegMaxLoopA ;note that pk width is in y0
_EndSlopeA
P:01B6  044913      lua    (r1)+n1, r3
                           ;make r3 point to the end of the sampled peak buffer
P:01B7  045D15      lua    (r5)+, r5
```

```
P:01B8  56D300    move   X:(r3)-, a    ;last point to A
P:01B9  44E300    move   X:(r3), x0    ;second last point to x0
P:01BA  655A44    sub    x0, a  r5, X:(r2)+  ;save data count at max.
P:01BB  455A00    move   x1, X:(r2)+         ;save max. value
P:01BC  465A00    move   y0, X:(r2)+         ;save data count at pk end
P:01BD  565A13    clr    a      a, X:(r2)+   ;save slope at pk end
                         ;clr a, if it is a neg. pk, make pha count 0
P:01BE  0A0281    jclr   #PK_SAMPLING_SIGN_A, X:PkSamplingSt,
        0001C2           _CheckFullA         ;if negative pk, skip PHA
P:01C0  0BF080    jsr    PHA    ;calculate MCA channel number, be sure the
        000211           ;Max. is in x1. when it returns the channel number
                         ;in A. Other register: r3 and B.
_CheckFullA              ;check pk buffer full
P:01C2  565A00    move   a, X:(r2)+    ;save PHA channel number
P:01C3  044E16    lua    (r6)+n6, r6  ;update pk buffer counter
P:01C4  620800    move   r2, X:PkBufferWriteA
                         ;save pk buf. write pointer
P:01C5  44F400    move   #>PEAKDBFSIZE, x0
        000080
P:01C7  22CE00    move   r6, a
P:01C8  560945    cmp    x0, a  a, X:PkBuffCntA     ;save pk buffer counter
P:01C9  0AF0A9    jlt    _ChannelB
        0001CC
P:01CB  0A0124    bset   #PEAKDBFAFULL, X:BuffStatus
_ChannelB
P:01CC  0A028A    jclr   #PK_SAMPLING_FINISH_B, X:PkSamplingSt, _Ret
        000210
P:01CE  6A8800    move   Y:PkBufferWriteB, r2
                         ;r2 is pk buffer writer pointer
P:01CF  63F400    move   #>PK_SAMPLE_START, r3
        007000           ;r3 is pk sampled buffer read pointer
P:01D1  6E8900    move   Y:PkBuffCntB, r6   ;r6 is pk buffer counter
P:01D2  76F400    move   #>8, n6
        000008
                ;n6 is the number of parameters per pk calucalate start slope
P:01D4  4EDB00    move   Y:(r3)+, y0  ;first point to y0
P:01D5  5EE300    move   Y:(r3), a
                         ;second point to A, note that r3 is not incremented
P:01D6  4C8654    sub    y0, a  Y:PkStartHi16B, x0
P:01D7  4C5A00    move   x0, Y:(r2)+
                ;calculate the start slope and save the, start point high 16 bits
P:01D8  4C8500    move   Y:PkStartLo16B, x0
P:01D9  4C5A00    move   x0, Y:(r2)+  ;save start point low 16 bit
P:01DA  5E5A00    move   a, Y:(r2)+
                         ;save start slope find peak max. time at max.
P:01DB  64F400    move   #>2, r4
        000002           ;r4 here is a counter
P:01DD  229500    move   r4, r5        ;r5 keeps the count at max.
P:01DE  4C8700    move   Y:PkWidthB, x0
P:01DF  0A0289    jclr   #PK_SAMPLING_SIGN_B,X:PkSamplingSt, _NegMaxB
        0001EE
P:01E1  4FDB00    move   Y:(r3)+, y1  ;y1 keeps the max. value
P:01E2  5EDB00    move   Y:(r3)+, a   ;a keeps the present value
```

```
_PosMaxLoopB
P:01E3  16DB75      cmp    y1, a  a, x1  Y:(r3)+, a    ;compare the current to
                           ;the max., keep the current in x1 and update a
P:01E4  0AF0AF      jle    _PUpdateB
        0001E8
P:01E6  20A700      move   x1, y1
                           ;if the new data is larger than the present max.
P:01E7  229500      move   r4, r5 ;update the max.
_PUpdateB
P:01E8  045C14      lua    (r4)+, r4     ;update r4 counter
P:01E9  228F00      move   r4, b
P:01EA  22994B      eor    x0, b  r4, n1 ;check peak end, n1 keeps track of r4
        ;counter, which is used later to offset r3 to the end of sample peak
P:01EB  0E21E3      jne    _PosMaxLoopB ;note that pk width is in x0.
P:01EC  0AF080      jmp    _EndSlopeB
        0001F9
_NegMaxB
P:01EE  4FDB00      move   Y:(r3)+, y1   ;y1 keeps the max. value
P:01EF  5EDB00      move   Y:(r3)+, a    ;a keeps the present value
_NegMaxLoopB
P:01F0  16DB75      cmp    y1, a  a, x1  Y:(r3)+, a    ;compare the current to
                           ;the max., keep the current in x1 and update a
P:01F1  0AF0A1      jge    _NUpdateB
        0001F5
P:01F3  20A700      move   x1, y1
                           ;if the new data is smaller than the present max.
P:01F4  229500      move   r4, r5 ;update the max. and data count at max
_NUpdateB
P:01F5  045C14      lua    (r4)+, r4     ;update r4 counter
P:01F6  228F00      move   r4, b
P:01F7  22994B      eor    x0, b  r4, n1 ;check end point, n1 keeps track of r4
        ;counter, which is used later to offset r3 to the end of sample peak
P:01F8  0E21F0      jne    _NegMaxLoopB ;note that pk width is in x0
_EndSlopeB
P:01F9  044913      lua    (r1)+n1, r3
                           ;make r3 point to the end of the sampled peak buffer
P:01FA  045D15      lua    (r5)+, r5
P:01FB  5ED300      move   Y:(r3)-, a          ;last point to A
P:01FC  4CE300      move   Y:(r3), x0    ;second last point to x0
P:01FD  6D5A44      sub    x0, a  r5, Y:(r2)+  ;save data count at max.
P:01FE  4F5A00      move   y1, Y:(r2)+         ;save max. value
P:01FF  4C5A00      move   x0, Y:(r2)+         ;save data count at pk end
P:0200  5E5A00      move   a, Y:(r2)+          ;save slope at pk end
P:0201  20E513      clr    a      y1, x1       ;save the pk max. in x1
P:0202  0A0289      jclr   #PK_SAMPLING_SIGN_B, X:PkSamplingSt, _CheckFullB
        000206                                 ;if negative pk, skip PHA
P:0204  0BF080      jsr    PHA    ;calculate MCA channel number, be sure the
        000211             ;Max., is in x1. when it returns the channel number
                           ;in A. Other registers: r3 and B.
_CheckFullB                ;check pk buffer full
P:0206  5E5A00      move   a, Y:(r2)+    ;save PHA channel number
P:0207  044E16      lua    (r6)+n6, r6   ;update pk buffer counter
P:0208  6A0800      move   r2, Y:PkBufferWriteB
```

```
                    ;save pk buf. write pointer
P:0209  44F400      move   #>PEAKDBFSIZE, x0
        000080
P:020B  22CE00      move   r6, a
P:020C  5E0945      cmp    x0, a  a, Y:PkBuffCntB    ;save pk buffer counter
P:020D  0AF0A9      jlt    _Ret
        000210
P:020F  0A0125      bset   #PEAKDBFBFULL, X:BuffStatus
P:0210  00000C      _Ret   rts
;-----------------------------------------------------------------
PHA     ;calculate MCA channel number, be sure the Max. is in x1. when
        ;itreturns the channel number in A. Other register: r3 and B. PHA
        ;scaling table is in Y_mem starting from #PHATABLESTART
P:0211  578D00      move   X:ChannelNum, b     ;copy total PHA channel number
                    ;to b. Remember the PHA channel number must be 2 to the
                    ;power of n for proper sorting.
P:0212  63F42B      lsr    b      #>PHATABLESTART, r3 ;half the channel
        000100             ;number, r3 now is pointer to PHA scaling table
P:0214  21BB00      move   b1, n3
              ;copy the half of the PHA channel number to n3 to offset r3
P:0215  000000      nop
P:0216  204B2B      lsr    b      (r3)+n3      ;half the channel number offset
P:0217  5EE300      move   Y:(r3), a
        ;fetch a data from PHA scaling table and update the table pointer.
P:0218  060C00      do     X:QsortCyc, _PHAEnd
        000223
P:021A  21BB65      cmp    x1, a  b1, n3       ;compare the peak value in x1
        ;with the data from PHA scaling table, update PHA table pointer offset
P:021B  0AF0A1      jge    _HalfLeft     ;if the data in a is bigger than peak
        000221             ;value, the table point is to offset to the left.
P:021D  204B2B      lsr    b      (r3)+n3
P:021E  5EE300      move   Y:(r3), a
                    ;Otherwise offset it to the right and half the offset in b
P:021F  0AF080      jmp    _SortAgain
        000223
_HalfLeft
P:0221  20432B      lsr    b      (r3)-n3
P:0222  5EE300      move   Y:(r3), a
_SortAgain
P:0223  000000      nop
_PHAEnd
P:0224  200065      cmp    x1, a
                    ;last comparasion for normalizing the channel to the left
P:0225  0AF0AF      jle    _SaveIt
        000228
P:0227  045313      lua    (r3)-, r3
                    ;if a>x1, the channel decrease by 1 to make it left
_SaveIt
P:0228  226E00      move   r3, a  ;save the PHA channel number in A
P:0229  57F400      move   #>PHATABLESTART, b
        000100
P:022B  200014      sub    b, a
P:022C  00000C      rts
```

```
;-------------------------------------------------------------------------
TXADBFtoHI                     ;transmit channel A data-buffer to HI
P:022D  09F0B0      movep X:TTL_Set, Y:$FFF0 ;request host ints for data
        000010      ;tranfer note that 'bset and bclr should not be used here
_TX1                           ;they change sampling rate unexpectedly
P:022F  0AA984      jclr  #M_HF1, X:<<M_HSR, _TX1
        00022F             ;wait for host acknowledge
P:0231  09F0B0      movep X:TTL_Clear, Y:$FFF0
        000011             ;clr host ints request
_TX2
P:0233  0AA981      jclr  #M_HTDE, X:<<M_HSR, _TX2
        000233
P:0235  08F4AB      movep #>DSPAHITX, X:<<M_HTX
        000000                    ;send tranfer code to host
_TX3
P:0237  0AA981      jclr  #M_HTDE, X:<<M_HSR,_TX3
        000237
P:0239  08DC2B      movep n4, X:<<M_HTX      ;send number of words to host
P:023A  06DC00      do    n4, _TXEnd
        00023E
_TX4
P:023C  0AA981      jclr  #M_HTDE, X:<<M_HSR, _TX4
        00023C
P:023E  08DCAB      movep X:(r4)+, X:<<M_HTX ;data tranfer from raw DBA to HI
_TXEnd
P:023F  00000C      rts
;-------------------------------------------------------------------------
TXBDBFtoHI                     ;transmit channel B data buffer to HI
P:0240  09F0B0      movep X:TTL_Set, Y:$FFF0 ;request host ints for data
        000010      ;tranfer note that 'bset and bclr should not be used here
_TX1                           ;they change sampling rate unexpectedly
P:0242  0AA984      jclr  #M_HF1, X:<<M_HSR, _TX1
        000242             ;wait for host acknowledge
P:0244  09F0B0      movep X:TTL_Clear, Y:$FFF0
        000011             ;clr host ints request
_TX2
P:0246  0AA981      jclr  #M_HTDE, X:<<M_HSR, _TX2
        000246
P:0248  08F4AB      movep #>DSPBHITX, X:<<M_HTX
        000001             ;send tranfer code to host
_TX3
P:024A  0AA981      jclr  #M_HTDE, X:<<M_HSR, _TX3
        00024A
P:024C  08DC2B      movep n4, X:<<M_HTX      ;send number of words to host
P:024D  06DC00      do    n4, _TXEnd
        000251
 _TX4
P:024F  0AA981      jclr         #M_HTDE, X:<<M_HSR, _TX4
        00024F
P:0251  08DCEB      movep Y:(r4)+, X:<<M_HTX ;data tranfer from raw DBA to HI
_TXEnd
P:0252  00000C      rts
;============= MAJOR COMMANDS =============
```

```
;--------------------------- 0: Record ---------------------------------
Record                      ;recording process from SSI to HI
comment @    Not yet finished    @
P:0253  00000C     rts
;--------------------- 1: On-line MCA ---------------------------------
OnlineMCA
P:0254  0AA823     bset   #M_HF2, X:<<M_HCR   ;tell host: not ready to start
_Mca1
P:0255  0AA980     jclr   #M_HRDF, X:<<M_HSR, _Mca1
        000255
P:0257  0870AB     movep  X:<<M_HRX, X:CountLo16Max
        00001D                  ;max data count low 16 bit
_Mca2
P:0259  0AA980     jclr   #M_HRDF, X:<<M_HSR, _Mca2
        000259
P:025B  0870AB     movep  X:<<M_HRX, X:CountHi16Max
        00001E                  ;max data count hi 16 bit
P:025D  0BF080     jsr    InstallSSIInts
        00033C                  ;install SSI ISR
P:025F  0BF080     jsr    InstallHostStopInts
        000354     ;install host stop ISR, very important that the two ISR
                   ;install utils before InitFIFO otherwise it would not work
                   ;correctly, since they use r7 as address pointer.
P:0261  44F400     move   #>$8001, x0
                   ;the circular buffer starts at $8000, here initiate
        008001     ;its read pointer to $8000+1 to get rid of the first data.
P:0263  441A00     move   x0, X:FIFORead
                          ;initiate the buffer read pointer
P:0264  0BF080     jsr    InitFIFO
        000361                  ;set up the circular buffer
P:0266  0BF080     jsr    InitProcStatus
        00036F                  ;initiate ProcStatus reg.
P:0268  0BF080     jsr    InitBuffStatus
        000378                  ;initiate BuffStatus reg.
P:026A  0BF080     jsr    McaConst
        0003A5                  ;get MCA parameters
P:026C  0BF080     jsr    LoadPHATable
        000398                  ;get PHA table
P:026E  44F413     clr    a       #>PEAKDBFSTART, x0
        006000
P:0270  440800     move   x0, X:PkBufferWriteA
P:0271  4C0800     move   x0, Y:PkBufferWriteB
                          ;initiate pk buffer write pointers
P:0272  540900     move   a1, X:PkBuffCntA
P:0273  5C0900     move   a1, Y:PkBuffCntB    ;initiate pk buffer counter
P:0274  5E0200     move   a, Y:PkSamplingCr
P:0275  560200     move   a, X:PkSamplingSt
P:0276  0A0081     jclr   #CHANNELA, X:ProcStatus, _ChanB
        000279
P:0278  0A0260     bset   #PK_SAMPLING_CHANNEL_A, Y:PkSamplingCr
_ChanB
P:0279  0A0082     jclr   #CHANNELB, X:ProcStatus, _ChanBSkip
        00027C
```

```
P:027B  0A0268      bset   #PK_SAMPLING_CHANNEL_B, Y:PkSamplingCr
_ChanBSkip
P:027C  0AA803      bclr   #M_HF2, X:<<M_HCR  ;tell host: ready to start
_WaitLoop
P:027D  0AA980      jclr   #M_HRDF, X:<<M_HSR, _WaitLoop
        00027D
P:027F  084F2B      movep  X:<<M_HRX, b
P:0280  20000B      tst    b
P:0281  0AF0AA      jeq    _McaStart
        000284
P:0283  00000C      rts
_McaStart
P:0284  0BF080      jsr    StartSSIInts
        000348                ;start up SSI interrupt
_McaLoop
P:0286  0A01A1      jset   #FIFOEMPTY, X:BuffStatus, _McaExit
        0002A1                ;if the circular buffer is empty, exit
P:0288  0A0184      jclr   #PEAKDBFAFULL, X:BuffStatus, _PkBuffB
        000293                       ;if pk buffer is full
P:028A  64F413      clr    a      #>PEAKDBFSTART, r4
        006000                       ;tranfer it to host and reset
P:028C  748900      move   X:PkBuffCntA, n4    ;the buffer pointer and count
P:028D  0D022D      jsr    TXADBFtoHI
P:028E  540900      move   a1, X:PkBuffCntA
P:028F  47F400      move   #>PEAKDBFSTART, y1
        006000
P:0291  470800      move   y1, X:PkBufferWriteA
P:0292  0A0104      bclr   #PEAKDBFAFULL,X:BuffStatus
_PkBuffB
P:0293  0A0185      jclr   #PEAKDBFBFULL, X:BuffStatus, _PkBuffFin
        00029E
P:0295  64F413      clr    a      #>PEAKDBFSTART, r4
        006000
P:0297  7C8900      move   Y:PkBuffCntB, n4
P:0298  0D0240      jsr    TXBDBFtoHI
P:0299  5C0900      move   a1, Y:PkBuffCntB
P:029A  47F400      move   #>PEAKDBFSTART, y1
        006000
P:029C  4F0800      move   y1, Y:PkBufferWriteB
P:029D  0A0105      bclr   #PEAKDBFBFULL, X:BuffStatus
_PkBuffFin
P:029E  0D00A9      jsr    PkSampling
P:029F  0D0188      jsr    PkDescription
P:02A0  0C0286      jmp    _McaLoop
_McaExit
P:02A1  0A02C0      jclr   #PK_SAMPLING_CHANNEL_A, Y:PkSamplingCr,
        0002AB             _McaFlushB
P:02A3  568900      move   X:PkBuffCntA, a     ;check if the buffer count is 0
P:02A4  200003      tst    a
P:02A5  0AF0AA      jeq    _McaFlushB
        0002AB
P:02A7  64F400      move   #>PEAKDBFSTART, r4
        006000                       ;empty the pk data buffers
```

```
P:02A9  21DC00      move   a, n4
P:02AA  0D022D      jsr    TXADBFtoHI
_McaFlushB
P:02AB  0A02C8      jclr   #PK_SAMPLING_CHANNEL_B, Y:PkSamplingCr, _McaRet
        0002B5
P:02AD  5E8900      move   Y:PkBuffCntB, a     ;check if the buffer count is 0
P:02AE  200003      tst    a
P:02AF  0AF0AA      jeq    _McaRet
        0002B5
P:02B1  64F400      move   #>PEAKDBFSTART, r4
        006000
P:02B3  21DC00      move   a, n4
P:02B4  0D0240      jsr    TXBDBFtoHI
_McaRet
P:02B5  0AA823      bset   #M_HF2,X:<<M_HCR    ;signal host for completion
_McaWt
P:02B6  0AA983      jclr   #M_HF0,X:<<M_HSR, _McaWt
        0002B6
P:02B8  0BF080      jsr    ReportStatus
        0002E1
P:02BA  0AA803      bclr   #M_HF2, X:<<M_HCR
P:02BB  00000C      rts
;-----------------------------------------------------------------------
OfflineMCA
P:02BC  000000      nop
P:02BD  00000C      rts
;====================INIT=================
;Entry point for the driver. Initializes the driver, sets up the DSP, then
;waits in a "command-interpreter" loop.
INIT_PGM
P:02BE  05F439      movec #$300, sr
        000300                         ;clear SR, none but lvl 3 ints
P:02C0  08F4BE      movep #0, x:<<M_BCR
        000000                         ;set the BCR to zero
                           ;init SSI interface
                    ;1) send a zero to TX so that SSI is initialized.
P:02C2  20001B      clr    b
P:02C3  08C92F      movep b0, X:<<M_TX ;write 0 to SSI output reg
                    ;2)init the SSI interface as needed.
                    ;CRA is set for 16-bit word length, 2-frame network mode
        ;CRB is set for xmit/rcv enabled w/ rcv interrupts ONLY, network mode,
        ;synchronous mode, SC0 as output.
P:02C4  08F4AC      movep #$4100, x:<<M_CRA
        004100                 ;normal
P:02C6  08F4AD      movep #$BA04, x:<<M_CRB
        00BA04
;Set up PCC to enable the interface
P:02C8  08F4A1      movep #$1f8, x:<<M_PCC
        0001F8                 ;enable SSI
;set sample rate
P:02CA  548F00      move   X:<MODELATCH, a1
P:02CB  09CC30      movep a1, y:<<$FFF0      ;write mode latch
;================= command loop ==============
```

```
CMDLUP
P:02CC  05F439      movec  #$300, sr
        000300              ;clear SR, none but lvl 3 ints
P:02CE  0AA803      bclr   #M_HF2, X:<<M_HCR
P:02CF  0AA804      bclr   #M_HF3, X:<<M_HCR  ;clear polling flags for host PC
CMDwait
P:02D0  0AA980      jclr          #M_HRDF, X:M_HSR, CMDwait
        0002D0                           ;wait for data at host port
P:02D2  084C2B      movep  X:<<M_HRX, A1       ;get HRX data
P:02D3  218400      move   A1, X0 ;save a copy in X0
P:02D4  4C0000      move   x0, Y:CommandWord
                           ;save a copy of command in Y:CommandWord
P:02D5  45F400      move   #>$F, x1            ;mask the fcn number
        00000F                                 ;mask for 4 lsbits
P:02D7  200066      and    x1, a
P:02D8  540E00      move   a1, X:<FUNCTION
                           ;save the fcn #. Note the copy in X0
P:02D9  219100      move   a1, r1        ;set up pointer to fcn list entry
P:02DA  391200      move   #fList, n1    ;set up base of fcn list array
P:02DB  000000      nop
P:02DC  67E900      move   X:(r1+n1), r7
                           ;load the address of the subroutine
P:02DD  0AA803      bclr   #M_HF2, X:M_HCR    ;clr HF2, used as completion
                           ;flag to Host (PC) execute the routine, note fcn
                           ;code is in X0 for commands that need it.
P:02DE  0BE780      jsr    (r7)  ;call the specific command
P:02DF  0C02D0      jmp    CMDwait       ;start again!
;================ SIMPLE COMMANDS ============
NULL                                     ;do nothing subr
P:02E0  00000C      rts
;---------------------------------------------------------------------------
ReportStatus                             ;report status to host
_RSE
P:02E1  0AA9A3      jset   #M_HF0, X:<<M_HSR, _RSE
        0002E1              ;wait host to signal start
_RSA
P:02E3  0AA981      jclr   #M_HTDE, X:<<M_HSR, _RSA
        0002E3
P:02E5  08F0AB      movep  X:ProcStatus, X:<<M_HTX
        000000             ;process status register
_RSB
P:02E7  0AA981      jclr   #M_HTDE, X:<<M_HSR, _RSB
        0002E7
P:02E9  08F0AB      movep  X:BuffStatus, X:<<M_HTX
        000001              ;buffer status register
_RSC
P:02EB  0AA981      jclr   #M_HTDE, X:<<M_HSR, _RSC
        0002EB
P:02ED  08F0AB      movep  X:CountHi16, X:<<M_HTX
        00001C              ;data count high 16 bit
_RSD
P:02EF  0AA981      jclr   #M_HTDE, X:<<M_HSR, _RSD
        0002EF
```

```
P:02F1  08F0AB      movep X:CountLo16, X:<<M_HTX
        00001B
P:02F3  00000C      rts
;------------------------------------------------------------------------
SampleRate   ;get sample rate from command word and set the sample rate
P:02F4  54F400      move   #$F00000, a1
        F00000             ;mask for sample rate data
P:02F6  200046      and    x0, a ;keep only bits 23..20 of command data
P:02F7  540F00      move   a1, X:MODELATCH     ;save it
P:02F8  09F0B0      movep  X:MODELATCH, y:$FFF0
        00000F             ;write mode latch
P:02FA  541100      move   a1, X:TTL_Clear
P:02FB  0A0F33      bset   #M_HIRQ, X:MODELATCH
P:02FC  568F00      move   X:MODELATCH, a
P:02FD  561000      move   a, X:TTL_Set
P:02FE  0A0F13      bclr   #M_HIRQ, X:MODELATCH
P:02FF  00000C      rts
;------------------------------------------------------------------------
UpLoadMem                  ;upload the contents of memory from DSP to host
_ULMH
P:0300  0AA9A3      jset   #M_HF0, X:<<M_HSR, _ULMH
        000300             ;wait host to signal start
_ULMA
P:0302  0AA980      jclr   #M_HRDF, X:<<M_HSR, _ULMA
        000302
P:0304  08452B      movep  X:<<M_HRX, x1
                    ;mem. type; 0:X_mem,1:Y_mem, 2:P_mem
_ULMB
P:0305  0AA980      jclr   #M_HRDF, X:<<M_HSR, _ULMB
        000305
P:0307  08462B      movep  X:<<M_HRX, y0      ;start addr
P:0308  20D600      move   y0, r6
_ULMC
P:0309  0AA980      jclr   #M_HRDF, X:<<M_HSR, _ULMB
        000305
P:030B  08462B      movep  X:<<M_HRX, y0      ;size of mem to be uploaded
_ULMD
P:030C  0AA9A3      jset   #M_HF0, X:<<M_HSR, _ULMD
        00030C      ;wait host to signal start
P:030E  20AE00      move   x1, a
P:030F  200003      tst    a
P:0310  0AF0AA      jeq    _X_mem
        00031E                 ;case of X memory
P:0312  47F400      move   #>1, y1
        000001
P:0314  200074      sub    y1, a
P:0315  0AF0AA      jeq    _Y_mem
        000324                 ;case of Y memory
_P_mem
P:0317  06C600      do     y0, _P_memEnd
        00031C                 ;case of P memory
P:0319  07DE87      movem  P:(r6)+, y1
_ULME
```

```
P:031A  0AA981      jclr   #M_HTDE, X:<<M_HSR, _ULME
        00031A
P:031C  08C72B      movep  y1, X:<<M_HTX
_P_memEnd
P:031D  00000C      rts
_X_mem
P:031E  06C600      do     y0, _X_memEnd
        000322
 _ULMF
P:0320  0AA981      jclr   #M_HTDE, X:<<M_HSR, _ULMF
        000320
P:0322  08DEAB      movep  X:(r6)+, X:<<M_HTX
_X_memEnd
P:0323  00000C      rts
_Y_mem
P:0324  06C600      do     y0, _Y_memEnd
        000328
_ULMG
P:0326  0AA981      jclr   #M_HTDE, X:<<M_HSR, _ULMG
        000326
P:0328  08DEEB      movep  Y:(r6)+, X:<<M_HTX
_Y_memEnd
P:0329  00000C      rts
;-------------------------------------------------------------------------
ZeroMem
P:032A  44F413      clr    a      #>$FBFF, x0
        00FBFF
P:032C  66F400      move   #>$100, r6
        000100
P:032E  06C400      do     x0, _exit
        000331
P:0330  546600      move   a1, X:(r6)
P:0331  5C5E00      move   a1, Y:(r6)+
_exit
P:0332  66F400             move   #>$1000, r6
        001000
P:0334  57F400      move   #>$FFFF, b
        00FFFF
 _Loop
P:0336  075E8C      movem  a1, P:(r6)+
P:0337  000000      nop
P:0338  22C400      move   r6, x0
P:0339  20004D      cmp    x0, b
P:033A  0E2336      jne    _Loop
P:033B  00000C      rts
;================= UTILITIES ================
InstallSSIInts      ;install SSI rcv data handler at $000C and $000E and HC at
        ;$0026 for upldm isr, pointer to instruction to poke is passed in x0
P:033C  05F439      movec  #$300, sr
        000300
             ;be sure that ints are shut off, clear SR, none but lvl 3 ints
P:033E  44F400      move   #>SSIDataInPtr, x0
        000090
```

```
P:0340  209700      move   x0, r7 ;set up pointer
P:0341  240000      move   #0, x0 ;need a zero to make a NOP
P:0342  07E78C      movem  P:(r7), a1
P:0343  070C0C      movem  a1, P:$000C  ;install ISR pointer
P:0344  070D04      movem  x0, P:$000D  ;add NOP after it
P:0345  070E0C      movem  a1, P:$000E
                           ;install ISR pointer in 'exception' ints
P:0346  070F04      movem  x0, P:$000F  ;add NOP after it
P:0347  00000C      rts
;-----------------------------------------------------------------------
StartSSIInts               ;starts up ISRs
P:0348  200013      clr    a
                           ;send a zero to TX so that SSI is initialized.
P:0349  08CE2F      movep  a, X:<<M_TX  ;write 0 to SSI output reg
                           ;init interrupt priority levels, enable interrupts
P:034A  08F4BF      movep  #$3800, x:<<M_IPR
        003800             ;set SSI IPL to 1 in the IPR, set hostIPL at 2 for
                           ;upload data and disable DEGMON monitor.
P:034C  00FCB8      andi   #$FC, MR     ;clear bits 0 & 1 of MR to enable ints
P:034D  00000C      rts
;-----------------------------------------------------------------------
StopInts            ;stops SSI ISRs but dosen't stop HC ISRs for upload isr
P:034E  05F439      movec  #$300, sr
        000300             ;clear SR, none but lvl 3 ints
P:0350  08F4BF      movep  #$0C00, x:<<M_IPR
        000C00                          ;reset SSI IPL in IPR to 0
P:0352  00FEB8      andi   #$FE, MR     ;clear bit 0 of MR to enable HI ints
P:0353  00000C      rts
;-----------------------------------------------------------------------
InstallHostStopInts
P:0354  05F439      movec  #$300, sr
        000300
              ;be sure that ints are shut off, clear SR, none but lvl 3 ints
P:0356  44F400      move   #>HostStopPtr, x0
        0000A4
P:0358  209700      move   x0, r7 ;set up pointer
P:0359  240000      move   #0, x0 ;need a zero to make a NOP
P:035A  07E78C      movem  P:(r7), a1
P:035B  07240C      movem  a1, P:$0024  ;install ISR pointer
P:035C  072504      movem  x0, P:$0025  ;add NOP after it
P:035D  00000C      rts
;-----------------------------------------------------------------------
HostStop
P:035E  0D034E      jsr    StopInts
P:035F  0A0020      bset   #HOSTSTOP, X:<ProcStatus
P:0360  00000C      rts
;-------------------------------------------------------------------
InitFIFO                   ;initialize FIFO pointers and M-regs
P:0361  60F400      move   #>$8000, r0
        008000             ;init the circular buffer write pointer
P:0363  370000      move   #0, r7 ;init write advance counter
P:0364  05F420      move   #>FIFOSIZE-1, m0
        007BFF                    ;make r0 modulo of FIFOSIZE
```

```
P:0366  00000C      rts
;----------------------------------------------------------------------------
ZeroXYMem                 ;clear X and Y memory for PHA
P:0367  66F400      move  #>PHATABLESTART, r6
        000100
P:0369  20001B      clr   b
P:036A  060084      do    #1024, _ZXYMEnd
        00036D
P:036C  576600      move  b, x:(r6)
P:036D  5F5E00      move  b, y:(r6)+
_ZXYMEnd
P:036E  00000C      rts
;----------------------------------------------------------------------------
InitProcStatus                  ;init process status register
P:036F  200013      clr   a
P:0370  540000      move  a1, X:<ProcStatus
P:0371  0A00C5      jclr  #5, Y:<CommandWord, _IPSA
        000374
P:0373  0A0021      bset  #CHANNELA, X:<ProcStatus  ;set input channel A bit
_IPSA
P:0374  0A00C4      jclr  #4, Y:<CommandWord, _IPSB
        000377
P:0376  0A0022      bset  #CHANNELB, X:<ProcStatus  ;set input channel B bit
_IPSB
P:0377  00000C      rts
;----------------------------------------------------------------------------
InitBuffStatus                  ;init buffer status register
P:0378  200013      clr   a
P:0379  540100      move  a1, X:<BuffStatus
P:037A  00000C      rts
;----------------------------------------------------------------------------
RVfromHI                  ;receive data from HI to the circular buffer
P:037B  09F0B0      movep X:TTL_Set, Y:$FFF0  ;request host ints for data
        000010            ;tranfernote that 'bset and bclr should not be used
                          ;here they change sampling rate unexpectedly
_RV1
P:037D  0AA984      jclr  #M_HF1, X:<<M_HSR, _RV1
        00037D                  ;wait for host acknowledeg
P:037F  09F0B0      movep X:TTL_Clear, Y:$FFF0
        000011                  ;clr host ints request
_RV2
P:0381  0AA981      jclr  #M_HTDE, X:<<M_HSR, _RV2
        000381
P:0383  08F4AB      movep #>DSPHIRV, X:<<M_HTX
        000002                  ;send tranfer code to host
_RV3
P:0385  0AA981      jclr  #M_HTDE, X:<<M_HSR, _RV3
        000385
P:0387  08D82B      movep n0, X:<<M_HTX      ;send number of words to host
P:0388  0A0081      jclr  #CHANNELA, X:<ProcStatus, _ChanB
        000391
P:038A  06DC00      do    n4, _RVChanA
        00038E
```

```
 _RVD
P:038C  0AA980      jclr   #M_HRDF, X:<<M_HSR, _RVD
        00038C
P:038E  085CAB      movep  X:<<M_HRX, X:(r4)+
                    ;data tranfer from HI to the circular buffer, for chA
_RVChanA
P:038F  0A0101      bclr   #FIFOEMPTY, X:<BuffStatus
P:0390  00000C      rts
_ChanB
P:0391  06D800      do     n0, _RVChanB
        000395
_RVE
P:0393  0AA980      jclr   #M_HRDF, X:<<M_HSR, _RVE
        000393
P:0395  085CEB      movep  X:<<M_HRX, Y:(r4)+
                    ;data tranfer from HI to the circular buffer, for chB
_RVChanB
P:0396  0A0101      bclr   #FIFOEMPTY, X:<BuffStatus
P:0397  00000C      rts
;--------------------------------------------------------------------------
LoadPHATable
P:0398  64F400      move   #>PHATABLESTART, r4
        000100
 _PTab1
P:039A  0AA980      jclr   #M_HRDF, X:<<M_HSR, _PTab1
        00039A
P:039C  0870AB      movep  X:<<M_HRX, X:ChannelNum
        00000D                          ;ChannelNum must be 2 to the power
P:039E  060D00      do     X:ChannelNum, _PTab2
        0003A3                          ;of N, for proper PHA sorting
_PTab3
P:03A0  0AA980      jclr   #M_HRDF, X:<<M_HSR, _PTab3
        0003A0
P:03A2  08452B      movep  X:<<M_HRX, x1
P:03A3  4D5C00      move   x1, Y:(r4)+
_PTab2
P:03A4  00000C      rts
;--------------------------------------------------------------------------
McaConst
_MCon1
P:03A5  0AA980      jclr   #M_HRDF, X:<<M_HSR, _MCon1
        0003A5
P:03A7  0870AB      movep  X:<<M_HRX, X:NoiseHi
        00000A
_MCon2
P:03A9  0AA980      jclr   #M_HRDF, X:<<M_HSR, _MCon2
        0003A9
P:03AB  0870AB      movep  X:<<M_HRX, X:NoiseLo
        00000B
_MCon3
P:03AD  0AA980      jclr   #M_HRDF, X:<<M_HSR, _MCon3
        0003AD
P:03AF  0870AB      movep  X:<<M_HRX, X:QsortCyc
```

```
        00000C                         ;The max number of sorting cycles for
:03B1   00000C      rts
             ;PHA, it is equal to base 2 logorithm of ChannelNum minus 1.
;------------------------------------------------------------------------
                    END      INIT_PGM
0    Errors
0    Warnings
```