Technische Universiteit Eindhoven

Faculty of Electrical Engineering

ICS
Section of Information and Communication Systems

Master's Thesis:

An intelligent weight controller using Profibus

H.A.J. Kester

id. nr. 380381

Location	: Ellips BV, Eindhoven, The Netherlands
Coach	: Ir. J.P.C.F.H. Smeets
Supervisor	: Prof. Ir. M.P.J. Stevens
Period	: March 1997 - September 1997

ELLIPS

SUMMARY

In this document we will discuss the design of a weight controller that is part of a new fruit grading system. The weight controller must be able to measure fruit that is passing by in cups with a precision of one gram. The weight sensor is a standard load cell that uses the Wheatstone bridge principle. Up to eight lanes of cups must be processed, with speeds of twenty cups per second per lane. The results of the weight measurement must be send to a master computer over a Profibus (Process Fieldbus) connection using the Profibus protocol. Furthermore, the weight controller must have the ability to update its software via the same Profibus channel.

The weight controller will be composed with the following components:

- a bridge excitation circuit
- a bridge amplifier
- a multiplexed A/D-converter
- a Digital Signal Processor
- an 80C32 host-processor
- a Profibus interface
- an FPGA

Two weight methods will be described:

- ♦ filtering the bridge signal with a FIR-filter and averaging ten filtered samples
- ♦ extracting the weight from the damping ratio and oscillation frequency of the bridge signal

The designed weight controller is suitable for the new fruit grading system, although it still has to be tested in practice. The "averaging" weight method works on an Agra machine with a precision of 2 grams. Better mechanical behaviour of the machine might improve this precision. The method using damping ratio and oscillation frequency still has to be tested.

For future versions of the weight controller it is desirable to implement Profibus-DP and Profibus-DPE to ensure proper operation in other Profibus-DP networks. Furthermore, the two weight methods must be tested on data from a different machine than one from Agra.

TABLE OF CONTENTS

1. INTRODUCTION

The last step of the stroll through the Eindhoven University of Technology is not the easiest one. Each faculty obliges its students to perform graduation work for several months and to write a report about it. At the faculty of Electrical Engineering this period contains either six or nine months, depending on whether or not a student follows the five-year course. A student at this faculty can choose one out of five sections at which he can graduate.

The section *Information and Communication Systems (ICS)* focuses on the total design and implementation trajectory of both the hardware and the software of digital systems. It contains two research chairs: *Computer and Communication Systems* and *Design Technology.* This document describes graduation work at the Computer and Communication Systems group. Research in this area of digital information systems covers information systems and computing systems, digital telecommunication and datacommunication systems (switches and networks), digital processors, operating systems and integrated circuits. Each student of this group can choose between graduating at the university itself or graduating in a company. Graduating in a company has the advantage of gathering work experience and is therefore quite popular among students. The graduation labour described in this paper was done at a company called Ellips.

Ellips is specialised in designing electronic control systems for vision applications, including all relevant software. Images from industrial and natural products are processed in real time. Examples of these products are raw materials, fruit and vegetables. Dimensions, contour characteristics, position, colour and quality are calculated in order to classify these products in terms of control parameters for the production process. The electronic computer systems are specially designed for easy integration with existing fruit and vegetable sizing mechanics.

In the next chapters we will discuss the design of an intelligent weight controller that will be part of the next generation of fruit grading systems.

2. FRUIT GRADING SYSTEM OVERVIEW

The concept of the new fruit grading system is outlined in figure 2.1. The general idea is that maximum flexibility must be offered to any customer using this system. Therefore, the system's architecture includes communication to the outside world. This allows Ellips to remotely diagnose and service the system, even when the actual factory is somewhere abroad. For example: the software of any device attached to the system can be updated from anywhere in the world by using the modem connection.

The grading system uses sensors for different purposes all attached to one of two Profibuses. These are special fieldbuses that will be described in chapter 3. One bus is used for peripherals that require fast data-exchanges in short messages, approximately 100 messages per second, and the other one is intended for devices that will produce

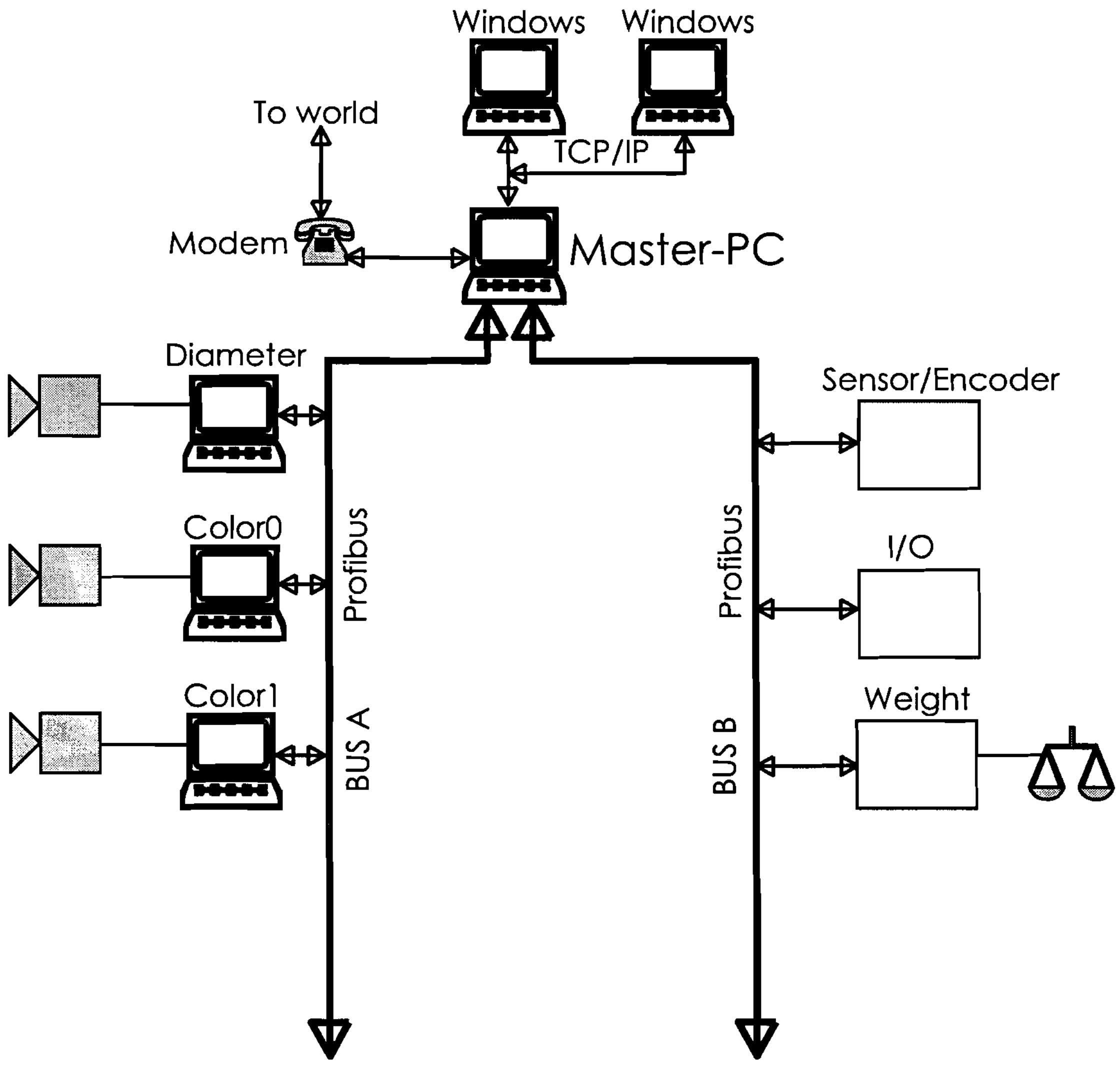


Figure 2.1: Fruit Grading Concept

data at a slower rate, for example 10 messages per second, but with more information. All devices are controlled by only one master computer.

This master computer is a Windows-based machine with the Profibus hardware in it. The master recognises each new device and configures it according to the user's specifications. During operation the master receives information of all devices attached to the buses to completely control the grading process. In the near future the following devices are possible [1]:

- weight controller - measures the weight of the cup and its contents
- diameter controller - measures the diameter of the fruit by means of a camera
- colour detector - detects the colour of the fruit inside the cup
- encoder - detects the position of the cups
- I/O-card - switches relays on and off to let the cups drop their contents

Since the diameter controller and the colour detector require only one data transfer for every cup but must be able to send complete images to the master computer, these devices are attached to the slower Profibus. The encoder is able to generate more than one message for each cup and thus must be connected to the fast bus. The I/O-card is used for dropping the contents of the cup, but is suited for any switching purpose. Therefore, switching at higher rates is also provided and the I/O-cards are also connected to the fast bus. The weight controller might be attached to both buses since it requires only one message per cup but does not need to send large messages. For the moment we will leave it attached to the fast Profibus connection.

Cups are passing by in lanes, with up to eight lanes for each machine. A unique number, the envelope number, will indicate every row of cups. Once a row of cups, maximal eight, passes a device, the master computer will indicate that device to start its measurement and will tell the device the envelope number of that row of cups. All devices will send their measurement results (such as weight, diameter or colour) to the master computer, accompanied by the envelope number. The master computer stores all the information of the cups of that envelope, until it has all the information needed to send the contents of a cup to a certain exit. This involves a simple and straightforward manner to sort the fruit.

Advantages of this new Profibus architecture are:

⇒ only two twisted-pair cables necessary instead of two wires for every device, often resulting in many kilometres of wire
⇒ higher speeds possible since the Profibus standard supports bit rates of 12 Mbit/s, which yields a machine speed of 20 cups per second
⇒ easy adding and removing of devices to the system
⇒ easy update of software of each device via the modem-connection and Profibus
⇒ addition of any third-party device designed for Profibus

3. PROFIBUS

3.1 The Profibus Standard

Serial fieldbuses are used today primarily as the communication system for exchange of information between automation systems and distributed field devices. Thousands of successful applications have provided impressive proof that use of fieldbus technology can save up to 40% in costs for cabling, commissioning and maintenance as opposed to conventional technology. Only two wires are used to transmit all relevant information (i.e., input and output data, parameters, diagnostic data, programs and operating power for field devices). In the past, incompatible vendor-specific fieldbuses were frequently used. Virtually all systems in design today are open standard systems. The user is no longer tied to individual vendors and is able to select the best and most economical product from a wide variety of products.

Some commonly used fieldbuses are CAN, WorldFIP, INTERBUS-S, P-NET and Profibus. Of these buses, only three are now standardised in Europe by the European Fieldbus Norm EN 50170: P-NET, WorldFIP and Profibus. This means that they will be accepted in all European countries. Several vendors have entered the fieldbus market and developed their own products for these standards. The Profibus standard is the most rapidly growing one, according to figure 3.1 (Source: CONSULTIC-Study).

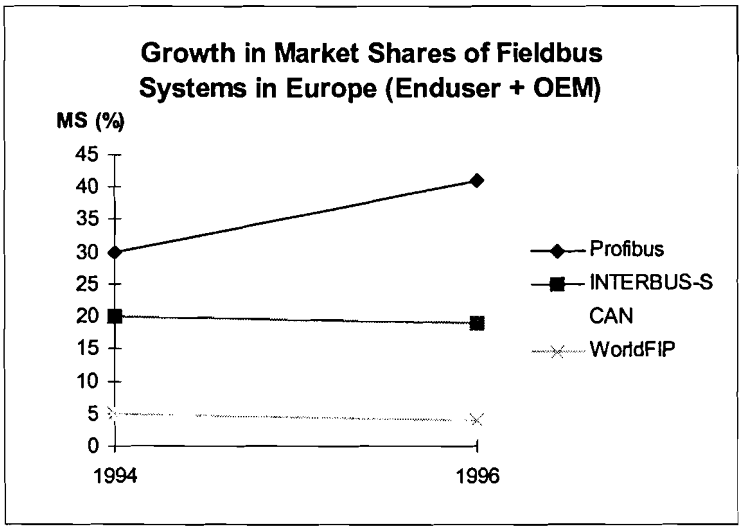


Figure 3.1: Market Shares In Europe

The fastest fieldbus is Profibus as well, which can operate at a maximum speed of 12 Mbit/s. Both the speed and the growing popularity of Profibus have been reasons for Ellips to choose this fieldbus for their new fruit grading system, which was described in chapter 2.

Profibus stands for Process Fieldbus and is developed in Germany. The first parts of the standards where published in 1991 in DIN 19 245 parts 1 and 2, followed by part 3 in 1994. As most communication protocols do, Profibus uses the Open Systems

Interconnection (OSI) model as proposed by the International Organisation for Standardisation (ISO). This model consists of seven layers, as shown in figure 3.2.

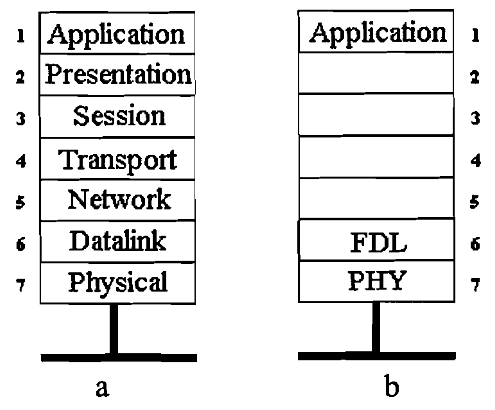


Figure 3.2: a) OSI-ISO 7-Layer Model b) Profibus OSI Model

Layer two is called the Fieldbus Data Link layer (FDL) in the Profibus version. In order to achieve high efficiency and throughput as well as low hardware and software costs, the layers 3 to 6 (Network, Transport, Session and Presentation) are left empty. A few relevant functions of these layers are realised in layer 2 or in layer 7 [2].

Three types of Profibus implementations are standardised:

• *Profibus-DP*	Decentralised Periphery, for high-speed data communication required in factory automation and building automation;
• *Profibus-FMS*	Fieldbus Message Specification, for object-oriented, general purpose data communication;
• *Profibus-PA*	Process Automation, meets the requirement of the process industry and offers applications for intrinsic safety and non intrinsic safety areas, as well as powering the field device over the bus.

The high speed of 12 Mbit/s of Profibus-DP is achieved by removing layer 7 in addition to the removal of layer 3 to 6. Thus, the DP version can be considered a standardised application of layer 2. Profibus FMS specifies several services in layer 7 and uses a maximum speed of 1.5 Mbit/s. Both Profibus-DP and Profibus-FMS uses Non-Return-To-Zero (NRZ) coding. Profibus-PA can achieve data rates of 31.25 kbit/s and uses Manchester coded signals, which implies a different layer 1 implementation, and a slightly different layer 2 to establish the safety requirements.

The Profibus network contains master stations and slave stations. A master is able to control the bus. This means that it may transfer messages without remote request when it has the right of access. Several masters can be connected to the same bus, and the one that has the right of access is the one that is holding the 'token'. The token circulates in a logical ring formed by the masters, see figure 3.3. If the system contains only one master, no token passing is necessary. This is a pure single master, multiple slaves system. In contrast to this a slave is only able to acknowledge a received message or to transfer data after a remote request. The minimum configuration comprises one master and one slave, or two masters. The maximum amount of devices that can be connected to the bus is 126. A device can be a master, a slave or a repeater.

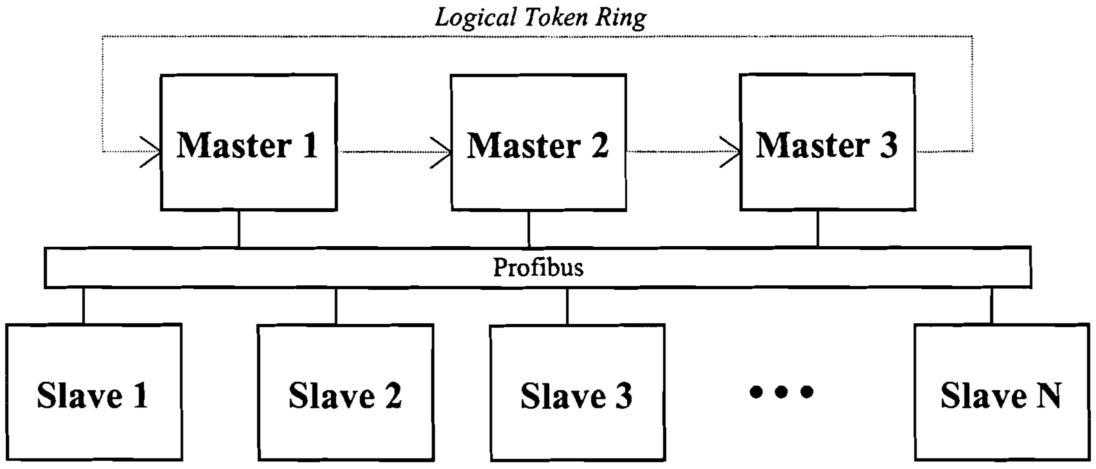


Figure 3.3: Profibus Concept

To support manufacturers making Profibus devices, the Profibus User Organisation was founded. This organisation has offices in numerous European countries, the United States, Australia, South Africa and Japan. The main goals are advancement of the technology, know-how transfer and protection of investments by influencing the standardisation process.

In order to assure that the Profibus network is reliable, even if the attached devices come from different manufacturers, the Profibus User Organisation has established a certification procedure for Profibus devices. Only devices that are accompanied by such a certificate may carry the Profibus trademark. The certification test is a standardised test procedure performed by experts, working in an accredited test lab. Experiences so far have shown that only uncertified products have caused system faults.

3.2 Physical Layer (PHY)

The first layer of the OSI model, the physical layer, describes how the data is send over the bus. This includes signal transmission, line requirements, data rates and the maximum number of stations. The Profibus standard uses a balanced line transmission corresponding to the US standard EIA RS-485.

Two line types are defined for Profibus: type A and type B. Type A is the modern version of the two, type B is the one described in the first DIN standards. New Profibus designs should be based on the modern line type A if possible. Both line types are shielded twisted pair cables. Fibre optics can be utilised as well, but this is beyond the scope of this text. The parameters for both line types are listed in table 3.1. The cable of type A is available from a number of manufacturers, for instance *Robert Bosch GmbH* or *Belden Wire + Cable.*

Table 3.1: Line Parameters		
Parameter	*Line A*	*Line B (Avoid if possible)*
Impedance (Ω)	135 to 165	100 to 130
Capacitance (pF/m)	< 30	< 60
Loop resistance (Ω/km)	< 110	-
Core diameter (mm)	> 0.64	> 0.53
Core cross section (mm^2)	> 0.34	> 0.22

The maximum line length depends on the data rate and whether or not repeaters are used. Without a repeater, the line may be as long as 1200 meters if the data rate does not exceed 93.75 kbit/s. By using three repeaters the line length can be increased to 4800 meters, more than sufficient for most applications. Table 3.2 lists some combinations of line lengths and data rates.

Table 3.2: Line Length versus Transmission Rate Without Repeaters							
Transmission rate in kbit/s	*9.6*	*19.2*	*93.75*	*187.5*	*500*	*1500*	*12000*
Type A	1200 m	1200 m	1200 m	1000 m	400 m	200 m	100 m
Type B	1200 m	1200 m	1200 m	600 m	200 m	70 m	-

The line should be terminated properly at both sides. For a type A line, three resistors should be used as in figure 3.4. Note that the data lines A and B have nothing to do with the cable types A and B as described above!

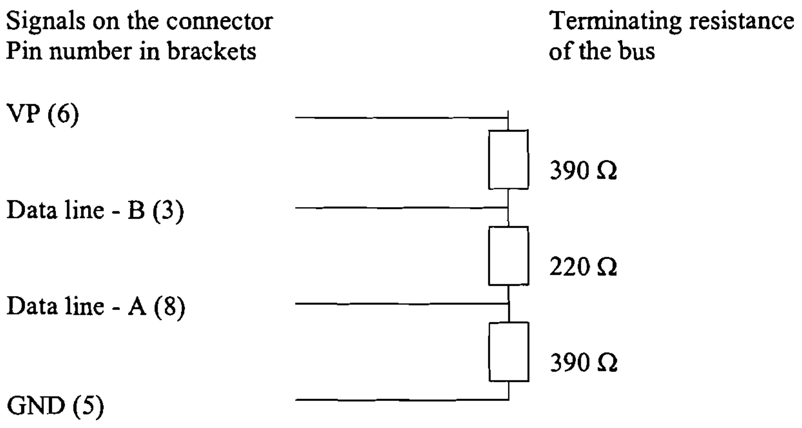


Figure 3.4: Line Termination

The plug connector is specified to be a 9-pin sub D connector. Four pins of this connector carry signals that are mandatory and must be supported by all devices. These signals are the same four as shown in figure 3.4. The other five signals are optional. All pin assignments are listed in table 3.3. The mandatory signals are printed in bold type.

Table 3.3: Pin Assignments Of The Bus Connector

Pin	*Signal*	*Designation*
1	Shield	Shield / Protective Ground
2	M24	Ground of the 24 V output voltage
3	**RxD/TxD-P (Data Line - B)**	**Receive data / transmission data positive**
4	CNTR-P	Control signal for repeaters (direction control)
5	**DGND**	**Data transmission potential (ground to 5 V)**
6	**VP**	**Supply voltage of the terminating resistance (5 V)**
7	P24	Output voltage 24 V
8	**RxD/TxD-N (Data Line - A)**	**Receive data / transmission data negative**
9	CNTR-N	Control signal for repeaters (direction control)

The user can provide 24 volts on pins 7 and 2 for connection of external operator control and maintenance devices that do not have their own power supply. The current carrying capability of the 24-volt connection must be at least 100 mA.

The line shield should be led to the protective ground of the device to prevent EMC interference from penetrating the device. This ground is usually the conductive housing of the device. Pins 4 and 9 are used for repeaters to indicate the direction of the signals.

3.3 Data Link Layer (FDL)

3.3.1 Token Procedures

As mentioned before and illustrated in figure 3.3, the masters in the Profibus network form a logical token ring. Only the master that holds the token is allowed to initiate a transmission. The token is passed from master to master in ascending numerical order of station addresses. The master with the highest address passes the token to the master with the lowest address. Each master knows its predecessor or Previous Station (PS), from which it receives the token. Furthermore each station knows its successor or Next Station (NS) to which the token is transmitted, as well as its own address or This Station (TS). Each master possesses a list of all masters in the system, the List of Active Stations (LAS). The LAS is generated after power on and updated or corrected, if necessary, later upon receipt of a token frame. If a master receives a token frame addressed to itself from the station that is marked in its LAS as Previous Station, it assumes that the token is passed to him. The station now owns the token and may start transmissions. If the token transmitter is not the registered PS, the addressee will assume an error and won't accept the token. Only after a second identical token frame is received, it will assume that the logical token ring is changed, update its LAS, grab hold of the token and start transmitting.

Once the master has finished its transmissions, it will pass the token to its successor, the NS. If the NS does not accept the token within a predefined time, the Slot Time, two retries will be attempted. If after the second retry the NS has still not responded, the active master station will assume the NS is no longer in the logical token ring and the token will be transmitted to the next station in the LAS. This procedure will continue until a master station accepts the token, or until there are no other masters left. In the latter case, the transmitter keeps the token or sends it to itself. If it finds an NS again in a later station registration, it tries again to pass the token.

Slaves are associated to one specific master and their addresses can be in the range between their master's address (TS) and the address of the NS. This address range is called the GAP and all the addresses in the GAP are represented in the GAP List (GAPL), except the address range between the Highest Station Address (HAS) and 127, which does not belong to the GAP. Another list, the Poll List, determines the order in which the slaves are polled. Once the master has received the token, it will start to poll the slave at the bottom of the Poll List. All slaves will be serviced in the order they appear in the Poll List.

Initialisation of the entire Profibus network after power on takes place if a master detects no bus activity within a certain time-out period. It shall claim the token, take it and start initialising. By transmitting two token frames addressed to itself, it informs any other master stations that it is now the only station in the logical token ring. Then it transmits a "Request FDL status" frame to each station in an incrementing address sequence, in order to register other stations. Stations responding with the message "slave station" or "master station not ready" are stored in the GAPL. The first master station that answers with "ready to enter logical token ring" will be registered as NS in the LAS and thus closes the GAP range of the master that holds the token. The token is passed to the master that was just found to allow that master to perform the same initialisation procedure.

3.3.2 *Frames*

Each Profibus frame consists of a number of frame characters called UART characters. The UART character (UC) is a start-stop character for asynchronous transmission, which is structured as in figure 3.5.

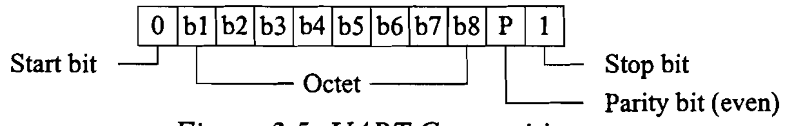


Figure 3.5: UART Composition

Five different frames are defined, each serving its own purpose.

- Frames of fixed length with no data field

SD1	DA	SA	FC	FCS	ED

- Frames with variable data field length

SD2	LE	LEr	SD2	DA	SA	FC	DATA_UNIT	FCS	ED

DATA_UNIT contains 1..246 octets

- Frames with fixed data field length

SD3	DA	SA	FC	DATA_UNIT	FCS	ED

DATA_UNIT contains 8 octets

- Token frame

SD4	DA	SA

- Short acknowledgement frame

SC

The abbreviations of the UARTs are listed in table 3.4.

Table 3.4: UART Abbreviations

	Value	*Description*
SD1	10H	Start Delimiter 1
SD2	68H	Start Delimiter 2
SD3	A2H	Start Delimiter 3
SD4	DCH	Start Delimiter 4
SC	E5H	Single Character
ED	16H	End Delimiter
DA	-	Destination Address
SA	-	Source Address
FC	-	Frame Control
FCS	-	Frame Check Sequence
LE	4..249	Octet Length
LEr	4..249	Octet Length repeated

The *LE* and *LEr* octets contain the number of octets in the variable data unit field plus the three previous octets (*DA, SA, and FC*). The *FCS* octet is used to check whether or not transmission was erroneous.

The address octets *DA* and *SA* in the frame header are composed as in figure 3.6.

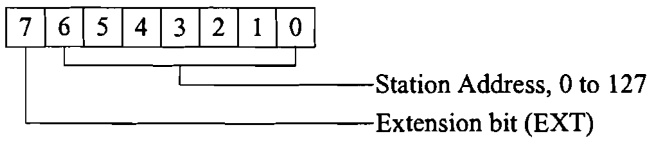


Figure 3.6: Address Octet

Address 127 is reserved as global address to broadcast messages to all stations. Furthermore, address 126 is reserved in some applications for new stations entering the Profibus network. The extension bit is used to indicate that an address extension follows immediately after the *FC* octet in the first octet of the data unit. Both the destination and the source address may have this bit set, so two octets might be occupied in the data unit by address extensions. An address extension has a structure as in figure 3.7.

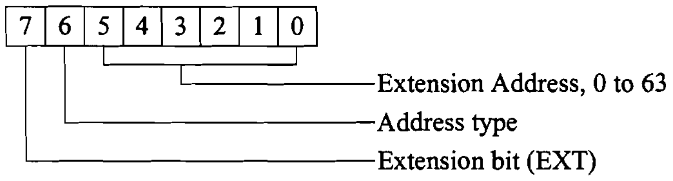


Figure 3.7: Address Extension Octet

If the type bit is set to zero, the address denotes a Link Service Access Point (LSAP) indicating a particular data transmission service. For the *DA* this is called a DSAP and for the *SA* this is called an SSAP. If the type bit is a one, the address denotes a regional or segment address used in bus systems with more than one segment. Once more, the extension bit indicates whether or not an extension to this address extension follows immediately after this octet.

The frame control octet *FC* in the frame header indicates the frame type, the function of the frame and control information to prevent loss and multiplication of messages, or the station type. Its structure is as in figure 3.8a and 3.8b.

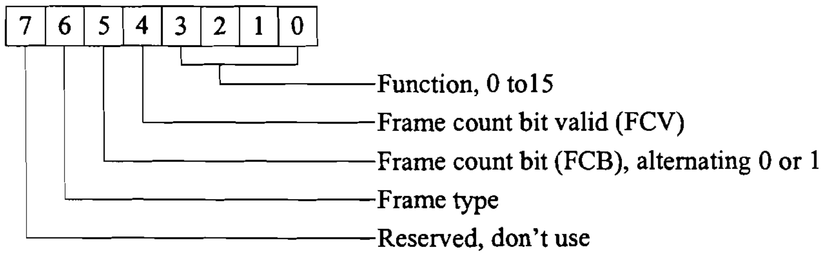

7 6 5 4 3 2 1 0
Function, 0 to15
Frame count bit valid (FCV)
Frame count bit (FCB), alternating 0 or 1
Frame type
Reserved, don't use

Figure 3.8a: Frame Control Octet for Request Frames

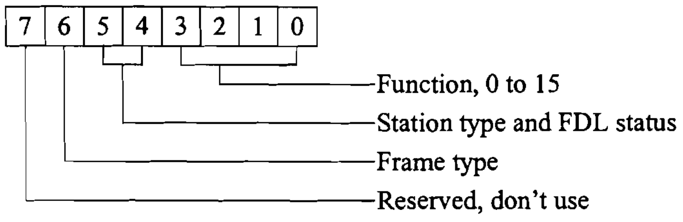


Figure 3.8b: Frame Control Octet for Response Frames

The reserved bit must really not be used. The *FCB* and *FCV* bits are used to prevent loss and multiplication of frames. Station type messages are listed in table 3.5.

Table 3.5: Station Type And FDL Status Messages

b5	*b4*	*Meaning*
0	0	Slave station
0	1	Master station, not ready to enter logical token ring
1	0	Master station, ready to enter logical token ring
1	1	Master station in logical token ring

The meaning of the function field depends on the frame type bit. The possible combinations are listed in table 3.6.

Table 3.6: Function Codes Of Frame Control Octet

Code	*Frame Type bit = 0 (Acknowledgement, Response Frame)*		*Frame Type bit = 1 (Request, Send / Request Frame)*
0	Acknowledgement positive	OK	Not used
1	Acknowledgement negative FDL user error	UE	Not used
2	Acknowledgement negative No resource for send data	RR	Not used
3	Acknowledgement negative No service activated	RS	Send data with acknowledge low
4	Reserved		Send data with no acknowledge low
5	Reserved		Send data with acknowledge high
6	Reserved		Send data with no acknowledge high
7	Reserved		Reserved
8	Response FDL data low (and send data OK)	DL	Not used
9	Acknowledgement negative No response FDL data (and send data OK)	NR	Request FDL status with reply
10	Response FDL data high (and send data OK)	DH	Reserved
11	Reserved		Reserved
12	Response FDL data low No resource for send data	RDL	Send and request data low
13	Response FDL data high No resource for send data	RDH	Send and request data high
14	Reserved		Request Ident with reply
15	Reserved		Request LSAP status with reply

3.3.3 Timing Specifications

Timing is vital in fieldbus applications, particular in our case of the fruit grading system. The system must be able to respond in a certain amount of time to commands issued by the master computer. This is called the System Reaction Time. We shall examine the reaction time for Profibus systems in this paragraph. We first define the Massage Cycle Time, T_{MC}, which is the time that elapses between two consecutive transmissions of a Profibus master station. Figure 3.9 illustrates the Message Cycle Time.

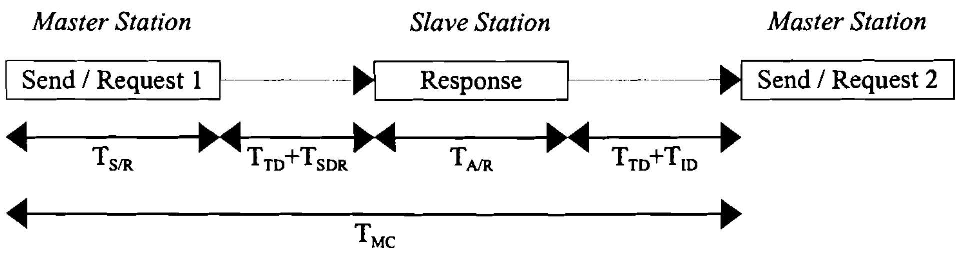


Figure 3.9: Message Cycle Time

The Message Cycle Time is therefore defined as:

$$T_{MC} = T_{S/R} + T_{SDR} + T_{A/R} + T_{ID} + 2 \cdot T_{TD} \tag{3.1}$$

where all times are in bit times ($T_{BIT} = 1$ / bit rate) and

$T_{S/R}$	=	Transmission time of the action frame = $a \cdot 11$ bits, a is the number of UART-characters to send
$T_{A/R}$	=	Transmission time of the response frame = $b \cdot 11$ bits, b is the number of UART-characters to send
T_{SDR}	=	Station Delay Time of Responders, time that elapses between request and response
T_{ID}	=	Idle Time, time that elapses between response and new request
T_{TD}	=	Transmission Delay Time, time that elapses on the transmission medium between transmitter and receiver when a frame is transmitted. $T_{TD} = \frac{\text{line length (m)} \cdot \text{bit rate (bit/s)}}{2 \cdot 10^8 (\text{m/s})} \text{bit}$

The System Reaction Time is determined from the Message Cycle Time. Suppose we have one master and *np* slaves in our system. Furthermore, the maximum amount of retries for a message is *mp*. The System Reaction Time is then

$$T_{SR} = np \bullet T_{MC} + mp \bullet T_{MC,RET} \tag{3.2}$$

where $T_{MC,RET}$ is the message cycle time for a retry. We shall assume that this is the same time as the normal Message Cycle Time.

In our fruit grading system, two timing values are important. We must be able to switch the relays within a certain amount of time to ensure that fruit will leave the system at the correct exit. The time between two consecutive polls of a relay is the same as the System Reaction Time if we assume that all relays are listed once in the master's Poll List. The second important timing is the time between two consecutive polls of the encoder card. To guarantee a certain degree of accuracy, the encoder must be polled more than once between two consecutive relay polls to give the exact conveyor position. This means that the encoder must be listed more than once in the Poll List. The desired specifications are that at a speed of 20 cups per second the system must be able to control the relays with an accuracy of 1/5 cup. This means that the System Reaction Time must be less than 10 milliseconds. The encoder precision must be 1/20 cup at the same speed, so the time between two consecutive encoder polls must be less than 2.5 milliseconds.

Suppose the Poll List is set up in such a way that the master polls *n* other slaves, then the encoder, then again *n* other slaves, the encoder again and so on, until all slave stations are serviced. Assume furthermore that the relays are polled with an SD2-frame containing 4 user bytes, thus 13 bytes total. The response of each relay is simply an acknowledgement with a SC-message of 1 byte. The weight controller has also 13 bytes in its action frame and uses a total amount of 29 bytes in the response SD2-frame. The encoder needs a 6-byte SD1-frame and a 13 bytes SD2-frame for its response. The Message Cycle Times for each device can then be calculated from equation (3.1).

The total Message Cycle Time will then be:

$$T_{MC,\ TOTAL} = nr \cdot T_{MC,RELAY} + T_{MC,WEIGHT} + \left\lceil \frac{np}{n} \right\rceil \cdot T_{MC,ENCODER} \qquad (3.3)$$

where

nr	=	the number of I/O-cards in the system
np	=	the total number of slaves in the system
n	=	the number of slaves polled between two encoder polls

The System Reaction Time T_{SR} can then be determined from (3.2). The time between two consecutive encoder polls is given by

$$T_{ENC} = \frac{T_{SR}}{\left\lceil \frac{np-1}{n} \right\rceil} \qquad (3.4)$$

Figures 3.10 and 3.11 show what the impact is of changing *n* with respect to the System Reaction Time and the encoder resolution for various bit rates, for a system containing one master, one encoder, one weight controller and 28 I/O-cards.

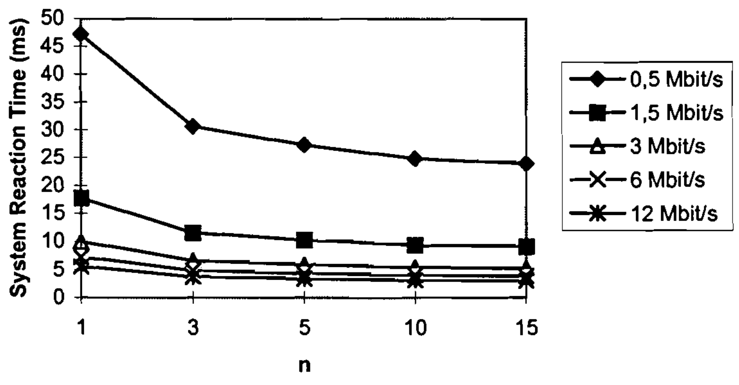


Figure 3.10: System Reaction Time versus n

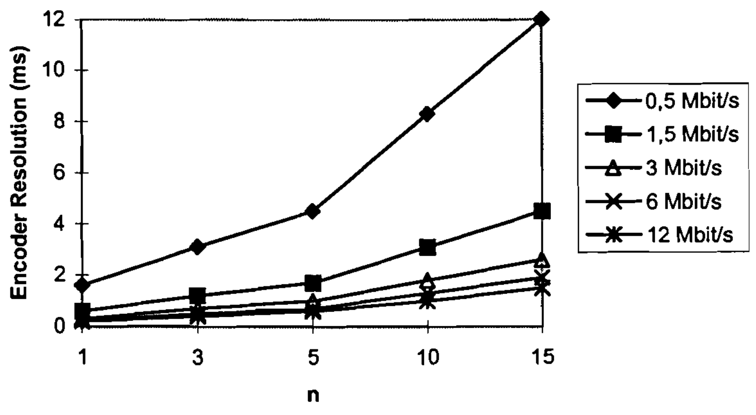


Figure 3.11: Encoder Resolution versus n

To meet the requirements of $T_{SR} \leq 10$ ms and $T_{ENC} \leq 2.5$ ms, the bit rate should be at least 1.5 kbit/s for this system, and the encoder should be polled after 8 other slaves, thus n=8. Tables with the calculated timing parameters, produced with Excel, can be found in Appendix B.

3.4 Profibus-DP

Since we are using decentralised peripherals in the fruit grading system, the Profibus-DP protocol seems a logical choice. It supports the highest bit rate of 12 Mbit/s and has several predefined services or service access points (SAPs):

• Default	Data exchange
• SAP54	Master-Master communication
• SAP55	Change station address
• SAP56	Read inputs
• SAP57	Read outputs
• SAP58	Control commands to a DP slave
• SAP59	Read configuration
• SAP60	Read diagnostic information
• SAP61	Transmit parameters
• SAP62	Check configuration

All these SAPS have predefined frame structures, which allows the use of any Profibus-DP slave from any manufacturer in the fruit grading system.

However, communications with the slaves still take place on a cyclic basis. In paragraph 3.3.3 we stated that it would be convenient to use a Poll List that contains the encoder device more than once to guarantee accurate timing. This is impossible in Profibus-DP. In future, some a-cyclic extensions will be added to the DP standard, referred to as DPE. As it is not a standard yet, only a few manufacturers support this feature for the master stations, resulting in astronomical prices. Furthermore, the solutions that are provided by these manufacturers are not quite elegant. For this reason, the first version of the fruit grading system will use the FDL layer directly. Future updates of the Profibus devices will support a-cyclic Profibus-DP.

4. WEIGHT CONTROLLER

4.1 Desired Specifications

The goal is to design a new weight controller that is able to operate in the environment as was described in chapter 2. This means that all communications between the weight controller and the master computer must be established via the Profibus protocol. The weight controller must have some sense of intelligence, since the master will only tell the weight controller to start a measurement and asks for the result of the measurement afterwards. The sequence of operation is as follows:

1. the encoder reports the exact location of a cup to the master computer
2. the master decides whether or not the cup has arrived at the weight controller
3. if the cup is in the right position, the master tells the weight controller to start a measurement
4. the weight controller measures the desired cup
5. the master asks the weight controller to send the measurement data
6. the weight controller sends the acquired data to the master computer if it has finished the measurement

Steps 5 and 6 are repeated until the weight controller has finished measuring.

The measurements are done with a standard load-cell that uses the Wheatstone-bridge measurement technique. Though the load-cells are standard, each user lets them operate in a different manner. Some of them use heavy cups so that the cup-weight is large compared to the weight of the fruit. This means that the measurement range is large as well. In practice, the weight controller must be able to handle a weight between 0 and 10 kilograms.

Each measurement must be accurate. For sorting purposes the weight controller must be able to measure with a precision of 1 gram. This requirement is also necessary for economical reasons. If someone delivers an amount of apples of quality A and B, he will get a higher price for his A-quality fruit. Prices are related to the weight of the fruit. Every apple is sorted on its quality and measured on its weight. The total weight of A-quality apples is determined by the sum of each single measurement. If those individual measurements deviate from the true value, someone might get less money for his apples or the customer pays too much.

The controller must be able to handle as much as 20 cups per second on each line of cups, with a maximum of eight lines. This yields a maximum of 160 cups to be measured every second. Since the signals from the load-cells are weak (1 gram corresponds to a few microvolts) an amplifier must be used. To ensure accurate measurements this amplifier must have low temperature drift and very low noise distortion. The amplified signals must be sampled with a sample rate of 1 kHz for each line of cups. This means that 8000 samples will be taken every second and must be processed in an accurate way.

The weight controller must also provide +5V and -5V power lines for each load-cell. These power signals should be ultra stable to ensure the required precision of the measurements. If the voltages drift due to an increase in temperature, the weight controller will measure a different weight than before the increase in temperature.

4.2 Controller Concept

The diagram of the weight controller is shown in figure 4.1. Eight Wheatstone-bridges are shown, representing the eight load-cells.

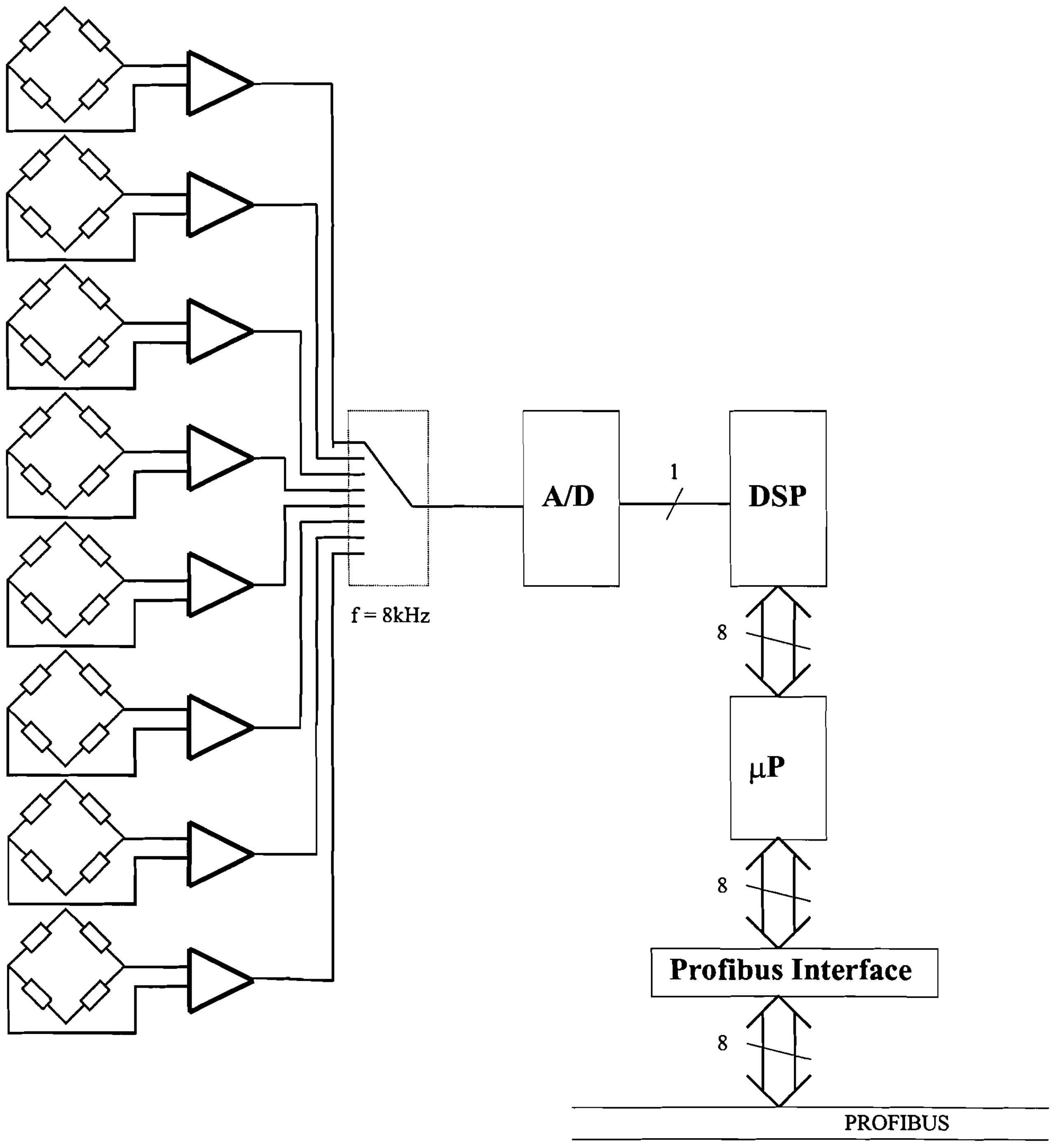


Figure 4.1: Weight Controller Diagram

Each bridge is connected to its own amplifier and all eight output-channels are connected to a multiplexer which selects one signal at a time. Note that the switching frequency is 8 kHz, since each bridge must be sampled at 1 kHz. An A/D-converter digitises the selected signal. To make sure that the measurement keeps the desired precision of 1 gram, the digital resolution must be at least 16 bits (see paragraph 5.3).

An ordinary microprocessor is not sufficient for calculating all 8000 measured values every second. The signals must be filtered and the actual weight must be calculated rapidly, exceeding the speed of a simple microprocessor. For these purposes, special digital signal processors (DSPs) are made. Their design and architecture are optimised for fast calculations and signal processing algorithms. After an A/D-conversion has taken place, the 16 bits of data are transferred serially to the DSP. If a sufficient amount of data is received, the DSP calculates the weight of the selected weight bridge. To pass the data to the master computer via the Profibus, an ordinary microprocessor is used. This processor communicates with the DSP via an 8-bit wide bus, called the Host Interface Port (HIP). The HIP allows the DSP to send status messages and weight data to the processor, while the processor can give commands to the DSP. In this configuration, the microprocessor is called the host and the DSP the slave.

The host processor takes care of the communication with the master computer. In order to do this, it communicates with the Profibus interface, which will take care of layer 1 and partly layer 2 of the Profibus protocol. The host has its own memory from which it can boot and in which it can store the programs running on the DSP.

An example of the complexity of the communication is the updating of the software of the DSP by someone in another country. Let's assume that the weight controller is connected to a fruit grading system in Italy and the new software is released in The Netherlands. First, the software is transferred from a computer in The Netherlands to the master computer connected to the Profibus network in Italy over the phone line using two modems. The master computer then invokes its Profibus hardware to set up communications with the Profibus interface of the host processor of the weight controller. Once the program is transferred to the host processor's memory, the host will send it to the memory of the DSP through the Host Interface Port. After the entire operation is completed, the DSP can execute the new program and continue its work.

In the next chapters, each component of the diagram will be discussed in detail.

5. Data Acquisition

5.1 Sensor

In 1856 Lord Kelvin discovered that applying strain to a wire shifted its resistance. This effect is repeatable, and is the basis for electrical output strain measurement. As mentioned before, the weight measurements in the fruit grading system are performed with standard load-cells that are using Wheatstone bridges as sensors. A diagram of such a bridge is shown in figure 5.1. If the bridge is in rest, the output voltage is trimmed to be exactly zero volts.

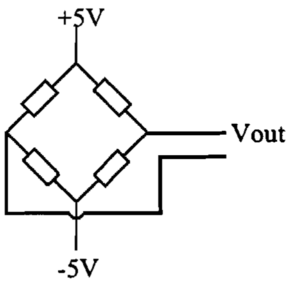


Figure 5.1: Wheatstone Bridge

The bridge contains four resistances of which at least one is a strain gauge. Applying strain to this gauge results in a different resistance for that gauge and thus for a difference in output voltage. Practical transducers must be trimmed for zero and gain, and compensated for temperature sensitivity.

In our fruit grading system, the load-cells are activated by the weight of the cups and the fruit inside the cups. The typical bridge output voltage is 2 to 4 mV per volt excitation. Since the maximum excitation voltage is 10 V for our load-cells, the maximum output voltage will be about 40 mV only. This maximum output voltage is reached when a weight of 10 kg is applied to the load-cell. This means that 1 gram will produce an output voltage of approximately 4 μV, indicating very stringent demands for the analogue circuitry with respect to noise distortion.

5.2 Signal Conditioning

Before the weak bridge signals can be converted to digital values they must be amplified to enhance the measurement's resolution. Nowadays, complete precision instrumentation amplifiers are fabricated by several manufacturers and sold in standard packages. Designers should take several device parameters into consideration to choose one of these amplifiers. These parameters are listed in table 5.1.

Table 5.1: Amplifier Parameters

Parameter	*Description*	*Value*
CMRR	Common Mode Rejection Ratio - If no weight is present on the load-cells, the voltage difference between the two output points is zero. However, the voltage difference between these points and ground voltage might be as much as 1 V. This results in an increase of the measured differential output voltage. The CMRR specifies the attenuation of the voltages present on the measurement points.	110 dB min
Drift	Due to a change in temperature the amplifier's output voltage will shift, resulting in measurement errors.	250nV/°C max
Noise	As mentioned before, one gram corresponds with only 4μV. Since the bandwidth is 1kHz, noise disturbance might introduce measurement errors of 1 gram.	Less than 4μV

No precision instrumentation amplifier can achieve all of the requirements stated in table 5.1. The LTC1250 from Linear Technology has insufficient noise performance and the INA128 from Burr-Brown drifts too much. However, the master computer can compensate for temperature drift if it keeps track of the initial weight of all empty cups. Measuring these cups once in a while when empty allows the master to adjust the weight results. Noise disturbance cannot be compensated since it cannot be distinguished from the amplified bridge signal. Thus, for our purpose, the INA128 suits best. With the gain set to 100 the maximum output voltage is measured to be 4.6 V and 1 gram corresponds to 460 μV.

For future updates, the design must have the ability of supporting both the implementations with the LTC1250 and the INA128, which are slightly different with respect to gain settings with resistances. In this way, any new precision amplifier based on either of these two components can be used without an entire redesign of the print layout. Figure 5.2 shows such a design, using replaceable 0 Ω resistances.

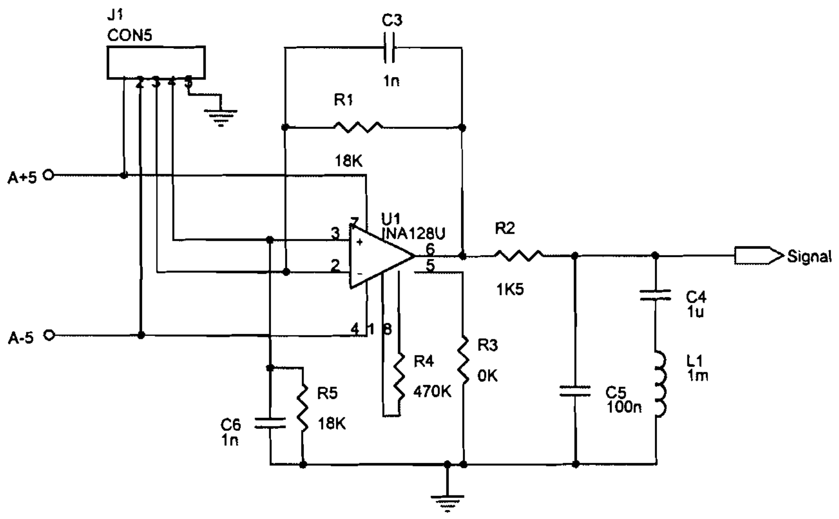


Figure 5.2: Amplifier Schematic

5.3 A/D-Conversion

The A/D-converter must be fast enough to convert analogue signals at a rate of 8 kHz. The requirement of a measurement precision of 1 gram involves a certain digital resolution to be obtained. Since most A/D-converters have an input range of -5 V to +5 V, the minimum amount of distinguishable digital levels is $\frac{10\ \text{V}}{460\ \mu\text{V}} = 21740$.

This involves a resolution of at least ${}^{2}\log 21740 = 14.5$ bits. To obtain a safe margin in processing the signals, a 16 bit A/D-converter is appropriate. This yields a total of 65536 different digital levels, which means that the least significant bit represents a value of 153 μV.

Note that in practice the measurement range is limited to several hundreds of grams, the weight of an average fruit element. Digital resolution might therefore be reduced significantly, implying fewer costs. This solution was used in the previous fruit grading system, not based on Profibus. However, for each single machine, a potentiometer had to be adjusted to trim the measurement range. As mentioned before, every customer has its own measurement method with different cup-weights and therefore different weight offsets. This potentiometer trimming is ineffective and often unreliable as well. For maximum flexibility, the measurement range is fixed to the full-scale range of the amplified bridge signal, thus requiring a 16-bit A/D-converter.

Each 16-bit dataword must be transferred to the digital signal processor. These processors are usually equipped with a serial port for communication purposes. The A/D-converter should therefor provide a serial port as well. Furthermore, the maximum integral nonlinearity (INL) must be less than 1.5 LSB. The differential nonlinearity (DNL) must be less than 1 LSB and is preferable zero. As with the precision instrumentation amplifiers, temperature drift must be small as well.

Only a few A/D-converters are suitable for our precision measurements. Since these A/D-converters are expensive and we would not like to be completely dependent of one single manufacturer, a converter that has the same functionality as and is pin-compatible with a second converter is preferable. Recently, Analog Devices announced its new AD977, a 200 kHz 16-bit A/D-converter with serial port, as a second source for the ADS7809 of Burr-Brown. Both converters are suited for our design with respect to INL, DNL and temperature drift. However, the AD977 is twice as fast and cheaper.

5.4 Bridge Excitation

To ensure an accurate non-drifting excitation voltage for all eight bridges, a precision voltage reference is used in combination with quality amplifiers. The amplifiers are used to regulate two power transistors. Since the bridges are standard 350 Ω load-cells operating at 10 V, the total supply current for the eight bridges is 0.23 A. The transistors must be able to supply these currents. Precision 5V voltage references are common devices as well as zero-drift amplifiers. Figure 5.3 shows the schematic for the bridge excitation circuitry.

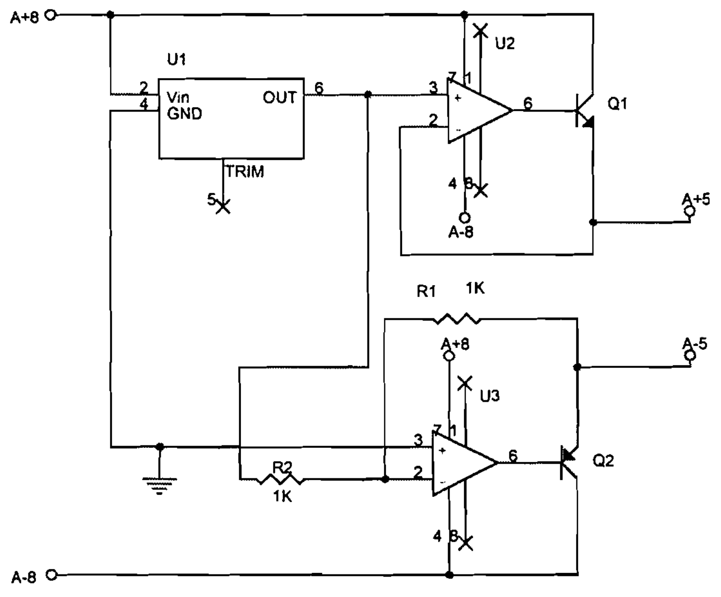


Figure 5.3: Bridge Excitation Schematic

The power supply of the weight controller must provide the two analogue voltages of –8 volts and +8 volts. All eight bridges are connected to the A+5 and A-5 lines. Some manufacturers of weight sensors use sense-wires to compensate for voltage drops in the cables to the bridge. A voltage drop will occur in both the positive and the negative supply cable, so the only result for our bridges will be that the gain is reduced. This is not important, since the weights have to be calibrated anyway.

6. SIGNAL PROCESSING

6.1 Digital Signal Processor

6.1.1 Texas Instruments versus Analog Devices

Today's leading manufacturers in Digital Signal Processors and Digital Signal Processor Tools are Texas Instruments and Analog Devices. A large amount of other companies have also tried to get their products on the market, but had either only special-functions DSPs, inferior DSP material or could simply not compete with their big brothers. Of the two leaders, Texas Instruments has the greatest market share at this time. During the past years, their product line increased steadily which is illustrated in figure 6.1.

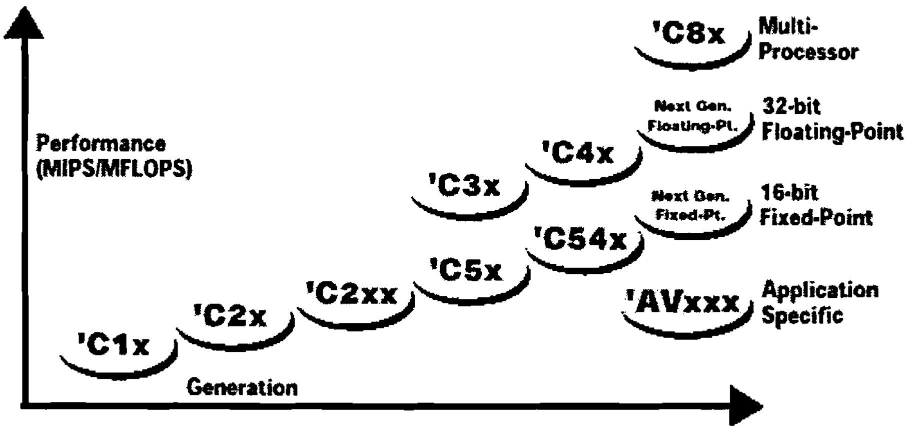


Figure 6.1: Texas Instruments DSP Product Line

The most advanced DSP generations have either 32-bit floating point arithmetic units or a multiprocessor core. Texas Instruments also provides an amount of software tools to develop and test DSP applications. These software tools are in general rather expensive, but since the prices of the DSPs are relatively low, the total development costs are reasonable if you are selling many DSP applications every year.

Analog Devices has based their DSPs on about the same architecture as Texas Instruments did. Their product line also features 16-bit fixed point as well as 32-bit floating point DSPs, growing steadily as well every year. A number of software tools is available for each DSP and at lower prices than those of Texas Instruments. Since the DSPs have a slightly higher price than those of Texas Instruments have, one can expect higher costs for big DSP projects.

Our product will be sold in amounts of approximately 100 each year and for that case, the costs won't differ very much. We must therefore compare the individual DSPs in terms of performance and ease of development. The basic requirements are that the DSP must be of the 16-bit fixed-point type and must be able to communicate with a host processor in a simple manner. Furthermore, it must have a serial port for data acquisition. Both manufacturers have developed fixed-point DSPs with serial ports and a host interface port (HIP). Texas Instruments offers the TMS320C57 as well as the TMS320C54x and Analog Devices has its ADSP-2171. We shall compare some differences between these devices in short.

We would like to boot our DSP through the Host Interface Port. Analog Devices has a predefined procedure for this, which simply lets the host tell the DSP how many instructions are going to be downloaded and expects the host to put the instructions in the proper registers of the HIP. This method is simplified by the HIP-splitter utility that puts all instructions in their proper position. Texas Instruments has a less elegant method of booting. After reset, the host must grab hold of the databus, write a byte somewhere in memory to indicate the manner in which the DSP should boot, release the databus again and must then fill a certain program memory space with instructions. After these instructions are downloaded, the DSP starts execution at the first instruction. This works, but the Analog Devices way is more graceful.

We are going to program in the C language, but we must also be able to generate assembly code for functions that require fast processing. If you take a look at a piece of assembly program of a Texas Instruments DSP, you cannot tell within a minute what the purpose of that particular piece of code will be. Since the assembly instructions are quite bizarre compared to those of other processors, most people refer to them as unreadable. Analog Devices excels in the readability of its assembly code since it uses algebraic expressions instead of ordinary assembly instructions. To illustrate this, two pieces of assembly code are shown below. Both are the cores of a FIR filter algorithm.

Texas Instruments code for the TMS320C54x [3]:
```
fir:          LD        #FIR_DP, DP
              STM       #K_FRAME_SIZE-1, BRC
              RPTBD     fir_filter_loop - 1
              STM       #K_FIR_BFFR, BK
              LD        *INBUF_P+, A
fir_filter:   STL       A, *FIR_DATA_P+%
              RPTZ      A, (K_FIR_BFFR - 1)
              MAC       *FIR_DATA_P+0%, *FIR_COFF_P+0%, A
              STH       A, *OUTBUF_P+
fir_filter_loop:
              RET
```

Analog Devices code for the ADSP-21xx [4]:
```
fir:          MR=0, MX0=DM(I0,M1), MY0=PM(I4,M5);
              DO sop UNTIL CE;
sop:          MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M5);
              MR=MR+MX0*MY0(RND);
              IF MV SAT MR;
              RTS;
```

Hence the Analog Devices DSPs require less time to master the assembly code instructions, reducing development time as well.

The assistance of local sales representatives in developing new applications is vital, especially for rather complex products as DSPs. Delivery times, availability of products and data sheets are the basic services provided by them. Furthermore, technical assistance is highly desired, since we do not have the expertise and experience in developing DSP products yet. The local representatives of Analog Devices master all these fields in a remarkable way. All questions are being answered in a minute and they have one person dedicated to technical questions regarding all Analog Devices DSPs. Texas Instruments failed to find proper representatives. Data sheets arrived in time, but the promised fax with sharp priced offers has not been seen yet and technical assistance is set to a poor level.

Thus, the easy-to-use Host Interface Port of the ADSP-2171, the clear assembly instructions and the outstanding support of Analog Devices local sales representatives make the difference to choose for this DSP instead of one of Texas Instruments.

6.1.2 ADSP-2171 Architecture Overview

Figure 6.2 is an overall block diagram of the ADSP-2171. The processor contains three independent computational units: the ALU, the multiplier/accumulator (MAC) and the shifter. The computational units process 16-bit data directly and have provisions to support multiprecision computations. The ALU performs a standard set of arithmetic and logic operations; division primitives are also supported. The MAC performs single-cycle multiply, multiply/add and multiply/subtract operations with 40 bits of accumulation. The shifter performs logical and arithmetic shifts, normalisation, denormalization, and derive exponent operations. The shifter can be used to efficiently implement numeric format control including multiword and block floating-point representations [5].

The internal result (R) bus directly connects the computational units so that the output of any unit may be the input of any unit on the next cycle.

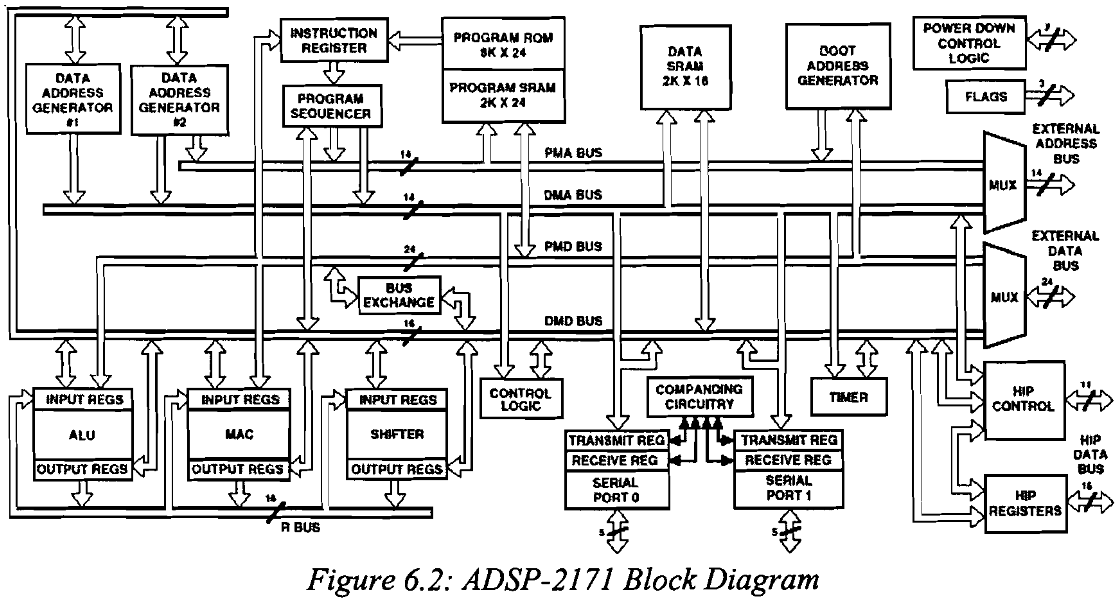


Figure 6.2: ADSP-2171 Block Diagram

A program sequencer and two dedicated data address generators ensure efficient delivery of operands to the computational units described above. The sequencer supports computational jumps, subroutine calls and returns in a single cycle. With internal loop counters and loop stacks, the ADSP-2171 executes looped code with zero-overhead; no explicit jump instructions are required to maintain the loop.

The two data address generators (DAGs) provide addresses for simultaneous dual operand fetches (from data memory and program memory). Each DAG maintains and updates four address pointers. Whenever the pointer is used to access data (indirect addressing), it is post-modified by the value of one of four possible modify registers. A length value may be associated with each pointer to implement automatic modulo addressing for circular buffers.

Efficient data transfer is achieved with the use of five internal buses.

- Program Memory Address (PMA) Bus
- Program Memory Data (PMD) Bus
- Data Memory Address (DMA) Bus
- Data Memory Data (DMD) Bus
- Result (R) Bus

The two address buses (PMA and DMA) share a single external address bus, allowing memory to be expanded off-chip, and the two data buses (PMD and DMD) share a single external data bus.

Program memory can store both instructions and data, permitting the ADSP-2171 to fetch two operands in a single cycle, one from program memory and one from data memory. Furthermore, the ADSP-2171 can fetch an operand from on-chip memory and the next instruction in the same cycle.

In addition to the address and data bus for external memory connection, the ADSP-2171 has a configurable 8- or 16-bit host interface port (HIP) for easy connection to a host processor. The HIP is made up of 16 data/address pins and 11 control pins. The HIP is extremely flexible and provides a simple interface to a variety of host processors, such as the Intel 8051 and the Motorola 68000. The host processor can initialise the ADSP-2171's on-chip memory through the HIP.

The ADSP-2171 can respond to eleven interrupts. There can be up to three external interrupts, configured as edge or level sensitive, and eight internal interrupts generated by the timer, the serial ports, the HIP, the powerdown circuitry and software.

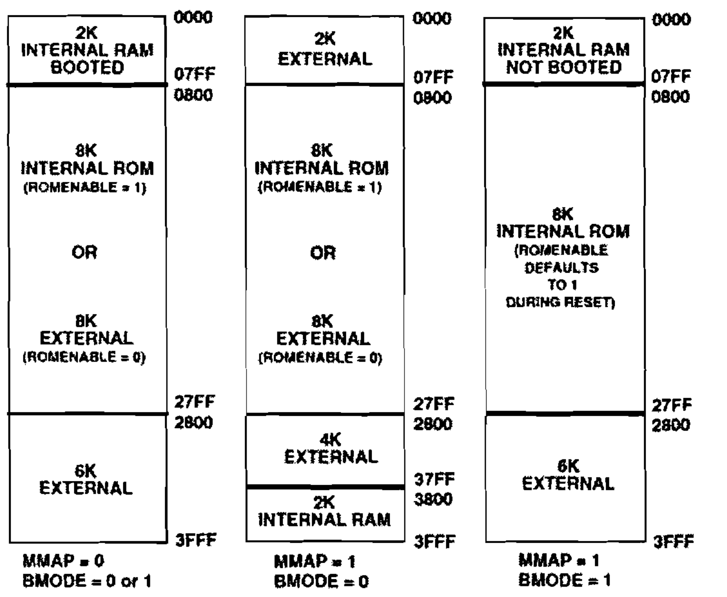
The two serial ports provide a complete synchronous serial interface with optional companding in hardware and a wide variety of framed or frameless data transmit and receive modes of operation. Each port can generate an internal programmable serial clock or accept an external serial clock.

The ADSP-2171 features three general-purpose flag outputs whose states can be simultaneously changed through software. These flag outputs can be used to signal an event to an external device. In addition, the data input and output pins on serial port 1 can be alternatively configured as an input flag and an output flag.

A programmable interval timer generates periodic interrupts. A 16-bit count register (TCOUNT) is decremented every *n* processor cycles, where *n-1* is a scaling value stored in an 8-bit register (TSCALE). When the value of the count register reaches zero, an interrupt is generated and the count register is reloaded from a 16-bit period register (TPERIOD).

6.1.3 Memory Maps

Program memory can be mapped in two ways, depending on the state of the MMAP pin. Figure 6.3 shows the different configurations. When MMAP=0, internal RAM occupies 2K words beginning at address 0x0000. In this configuration, the boot loading sequence is automatically initiated after a reset.



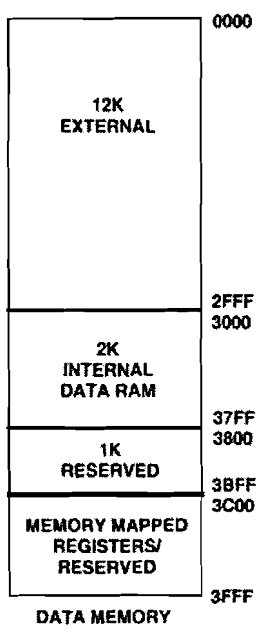


Figure 6.3: Program Memory Maps *Figure 6.4: Data Memory Map*

When MMAP=1, words of external program memory begin at address 0x0000 and internal RAM is located in the upper 2K words, beginning at address 0x3800. In this configuration, program memory is not loaded although it can be written to and read from under program control.

The BMODE pin determines the boot sequence. If BMODE=0 the ADSP-2171 boots through the data bus. If BMODE=1 the ADSP-2171 boots through the host interface port. In our application, we will use the MMAP=0, BMODE=1 configuration: booting through the host interface port.

The on-chip data memory RAM resides in the 2K words of data memory beginning at address 0x3000, as shown in figure 6.4. In addition, data memory locations from 0x3800 to the end of data memory at 0x3fff are reserved. Control registers for the system, timer, wait state configuration, host interface port, and serial port operations are located in this region of memory. The remaining 12K of data memory are external.

We shall use the MMAP=0, BMODE=1 configuration. Three 8-bit fast 32k SRAM chips can be used to implement both the 16k 24-bit program memory and the 16k 16-bit data memory. The remaining 8 kilobytes of memory are unused.

6.2 Software Architecture

6.2.1 Modular Approach

The DSP has a number of tasks to perform. It must be able to retrieve the incoming data, process it in some way and deliver the resulting weights at the host processor. In the next paragraphs we will discuss data retrieval and processing. Communicating with the host processor is an issue described in the next chapter.

Processing signals can be done in a number of ways. We might for instance simply filter the incoming signal and select an appropriate value to represent the weight. We can also take a number of filtered values and take the average of them to be the weight. A third method is not to filter the signal but to analyse the incoming waveform and try to recover the weight out of it by means of more intelligent algorithms. The processing task can therefore be implemented in more than one way. However, the acquisition of the data stream and communications with the host processor will be exactly the same for each method. This implies a software architecture that has the ability to replace the weight algorithm and has a fixed data-retrieval and host-communications part.

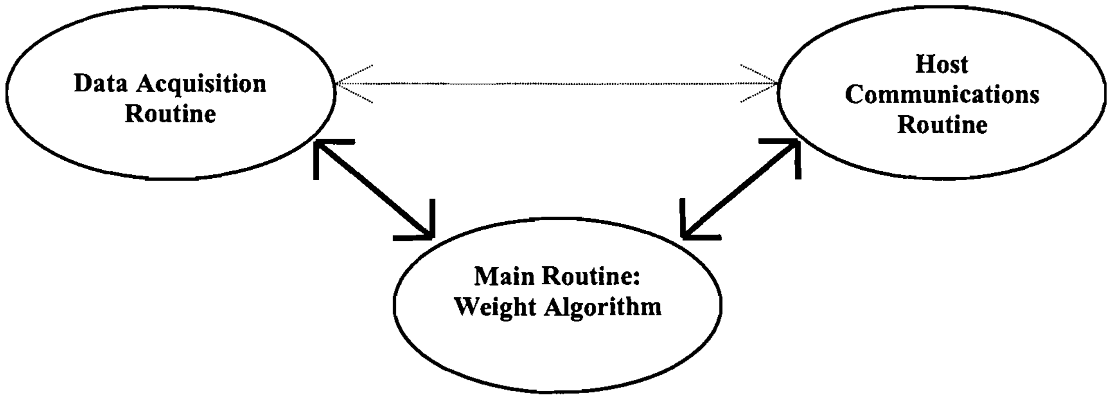


Figure 6.5: Software Architecture

Figure 6.5 illustrates this concept. The main routine performs the measurement task with data it receives from the data acquisition routine and sends and receives its instructions to the host via the host communications routine. Replacing the weight algorithm does not affect the other two routines. These routines can communicate with each other as well, for instance if the host processor asks for a curve of raw data values straight from the weight bridges. This can be done without notice of the main weight algorithm. A high degree of flexibility is assured in this way.

Communications between each routine is initiated by means of flags. If new data has been received by the data acquisition part, it sets a flag for the main routine. The main routine simply waits for this event to happen and processes the data in its own way. If the main routine has results to be transferred to the host, it sets data into a transmission queue and tells the host routine to send it.

6.2.2 Basic Runlevels

The software operates with a number of states or runlevels. Each runlevel has its own task and its own set of commands that can be send by the host. The number of runlevels depends on the weight algorithm but at least four runlevels are present, as shown in figure 6.6.
As can be seen, the four runlevels are:

- *Booted* - DSP has completed booting;
- *Gone* - DSP is running loaded program;
- *Init* - DSP is initialising parameters;
- *Measuring* - DSP is measuring weights.

The *Init* runlevel is for example used to set up the sample frequency, delay times, the channels that are going to be used, the filter length or filter constants. Each of these functions has its own command that can only be executed if the DSP is in the *Init* runlevel. If the host nevertheless sends one of these commands in a different runlevel, an error message will be returned. Some commands are not associated with a specific runlevel. These commands are general-purpose messages that need to be serviced at any time during operation. Table 6.1 shows an overview of the basic commands for every runlevel and table 6.2 of the general-purpose commands. A complete command overview can be found in Appendix F. Additional commands might be implemented for each weight-measuring algorithm.

Note that since only one command can be executed at the same time, the DSP will return an error message if the host sends a new command while the old one has not finished yet. There are a few exceptions to this rule, however. The commands that cause the DSP to switch to a lower runlevel are serviced regardless of the fact that an older command is in progress. This allows the host to abort all commands immediately, without having to wait until the DSP has finished them. An example of such a command is COM_STOP_MEASUREMENTS.

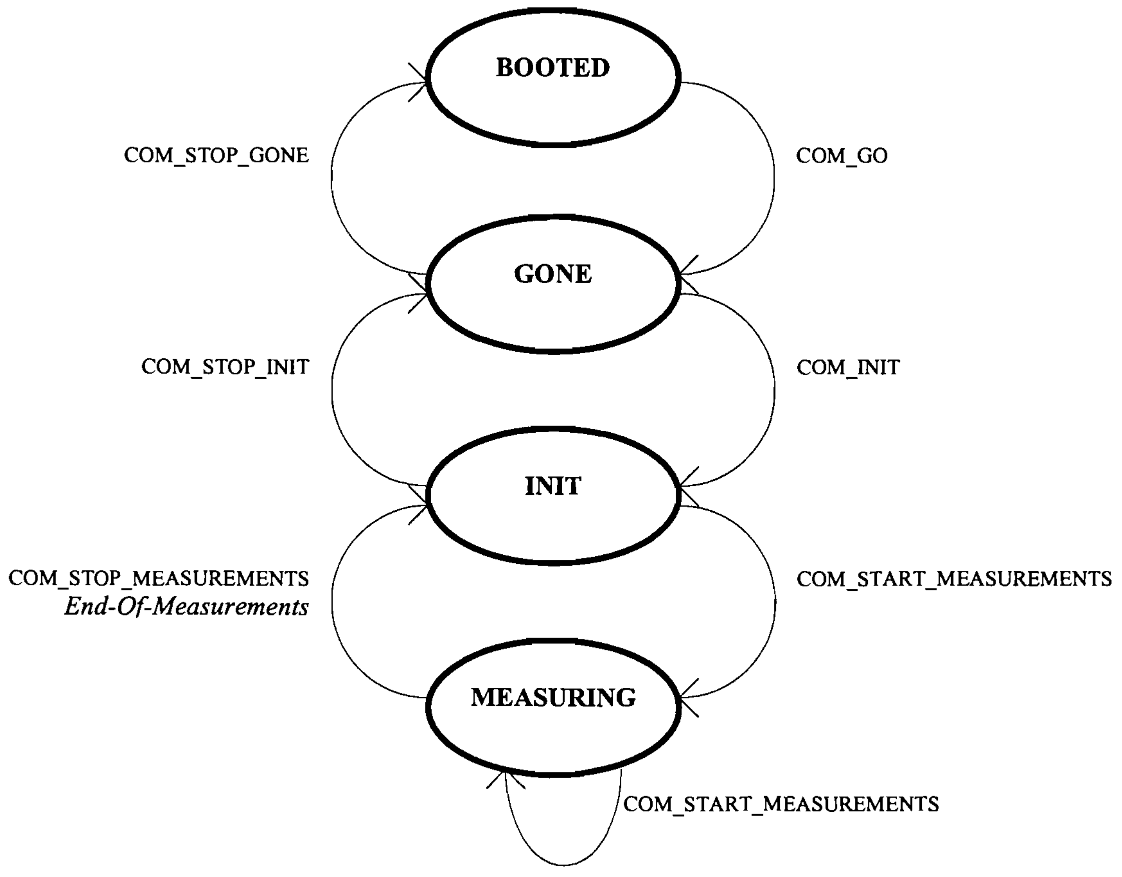


Figure 6.6: Basic Runlevels

Table 6.1: Basic Command Overview

Runlevel	*Command*	*Description*
Booted	COM_GO	Starts executing the loaded program.
Gone	COM_INIT	Switches to the initialisation runlevel.
	COM_GET_RAW_CURVE	Returns a curve of measured points straight from the A/D-converter. Parameters are the channel number, the number of points and the sample frequency.
	COM_STOP_GONE	Returns to the previous runlevel and stops measuring the raw curve.
Init	COM_SET_DELAY	Sets delay values for each channel to adapt to bridge misplacement errors.
	COM_SET_CHANNELS	Selects the channels that are going to be used for measurements.
	COM_SET_SAMPLE_FREQ	Sets the sample frequency of the A/D-converter for one channel.
	COM_START_MEASUREMENTS	Starts measuring on the selected channels.
	COM_STOP_INIT	Returns to the previous runlevel.
Measuring	COM_START_MEASUREMENTS	*See above*
	COM_STOP_MEASUREMENTS	Aborts all measurements immediately and returns to the previous runlevel.

Table 6.2: General Purpose Commands

Command	*Description*
COM_GET_RUNLEVEL	Reports the current runlevel.
NEW_DATA	Host or DSP has send new data through the Host Interface Port. *See chapter 7.*
HIP_ACK	Acknowledgement of receipt of command or data. *See chapter 7.*

6.2.3 Measurement Jobs

Figure 6.6 also indicates that in the *Measuring* runlevel a new measurement can be started while the other one was not finished yet. This is done by means of measuring jobs. Once a new measurement is started, a new job is initiated. This job exists as long as it takes to perform the measurement. After the measurement job is done, the result is send to the host processor and the job is killed. Once all jobs are killed, the program returns to a lower runlevel. The default maximum amount of jobs is four.

The structure of a job is defined to be:

```
typedef struct jobst
{
 uint16 active;           /* 0=inactive job, 1=active job */
 uint16 cup_number;       /* number of the measured cup   */
 uint16 timer[8];         /* counts down until delay time
                             has passed                   */
 ..                       /* additional data              */
} jobt;
```

Additional data may be added for each weight algorithm. An example of this will be given in paragraph 6.5. To kill a job, `active` is simply set to zero. The cup number indicates which cup is being measured, in order to be able to tell the master computer which cup we determined the weight of. The `timer` array is used to set a delay for each channel. This is done to take into account that not all eight possible bridges are set exactly in line. The cup in lane 1 might reach the bridge sooner than the cup in lane 2, so that the delay value for channel 2 must be set to a higher value.

Whatever the weight algorithm is, it must implement two functions:

```
void new_data();
void init_new_job(uint16 job_number);
```

The `new_data` function must process the data that was retrieved from the channel number indicated by a flag. It must check for all active jobs whether or not the transferred data is of interest, process it and send the result to the Host Communications Routine if the job is done. Once the job is done, it must be killed to allow a new measurement to use it.

The `init_new_job` function must fill in the additional data variables in the job structure. This might be additional counters or the sum of several weight samples. Note that this function may be left empty if no additional data is required.

6.3 Data Transfer from A/D-converter

The ADS7809 A/D-converter has several possibilities of transferring the acquired data to the DSP [6]. Data will always be clocked out serially in one of two formats. These formats are Straight Binary or Binary Two's Complement. Since the DSP uses Binary Two's Complement internally during normal operation, this output format is chosen. The data bits are clocked out at a rate specified by either the internal data clock or an externally connected data clock. The internal clock has a fixed rate of 2.3MHz while the external clock frequency may range from 0.1 to 10 MHz. For maximum flexibility and speed, we choose for an external clock and let the DSP generate this clock signal. The DSP is able to produce a clock signal of as high as one-half its processor's clock rate, so the ADS7809 will be the limiting factor.

With an external clock selected, we have two choices left for the actual transfer. The first one is to read the results after a conversion has taken place and the second one is to read the results during conversion. This last method will produce output data from the previous conversion while the next conversion takes place. Since this method is more complicated and only necessary for extremely high data throughput, we choose for the first option.

The required timing diagram is shown in Appendix C. A conversion is initiated by lowering the $R/\overline{C}$ signal first, followed by the $\overline{CS}$ signal for at least 40ns. The ADS7809 will now make $\overline{BUSY}$ low, indicating that the conversion is in progress. This will take maximally 10μs after which the $\overline{BUSY}$ signal is made high again. The conversion is now completed and the data transfer can take place. The $R/\overline{C}$ signal must first be set to a high level again, followed by lowering the $\overline{CS}$ signal. The ADS7809 will now send a synchronisation pulse to the DSP on the SYNC line. This pulse is active high so that the DSP must be set to use active high synchronisation pulses as well. This can be accomplished by setting bit 6 of the Sport 0 Control Register to zero. After the DSP has received the synchronisation pulse, it will receive the entire 16-bit dataword automatically. This is true only when SLEN, the lower 4 bits of the Sport 0 Control Register, is set to the binary sequence 1111. The data transfer takes place while the processor will continue to operate at full speed. After receiving the last bit, the 16-bit word is stored internally and an interrupt is given. An interrupt routine must take care of the proper storing of this dataword in the right memory location, so that the weight algorithms can process it at the earliest convenience.

6.4 Signal Analysis

As previously mentioned the load cells are Wheatstone bridges. This means that we can model these load cells as a damped mass-spring system. Such a system is drawn in figure 6.7.

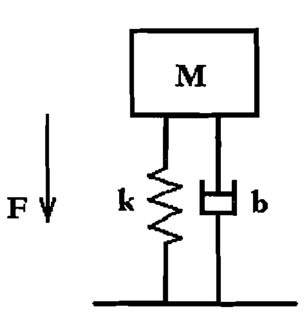


Figure 6.7: Model Of Damped Mass-Spring System

This model consists of a mass resting on a spring with spring constant *k* and a damper with friction constant *b*. The force F acting on the mass is given by Newton's law for motion [7]:

$$F = -m \cdot \ddot{y}$$

The same force F is equal to:

$$F = b \cdot \dot{y} + k \cdot y$$

This yields to the equation:

$$\ddot{y} + \frac{b}{m} \cdot \dot{y} + \frac{k}{m} \cdot y = 0$$

The Laplace transform of this equation produces the transform function H(s) that can be used to study the system's response to any excitation.

$$H(s) = \frac{1}{s^2 + \frac{b}{m} \cdot s + \frac{k}{m}}$$

This is the transfer function of a second order system. Therefore, the system's response will have all the characteristics of a second order system like overshoot, oscillation and damping.

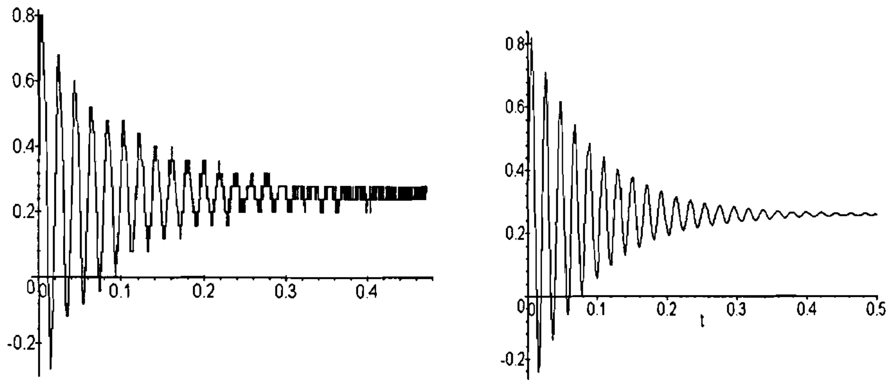
Suppose we would knock on the weight bridge for a very short time. This knock would be just like an excitation of the bridge with a Dirac delta function. The response of the system would be the solution to the homogeneous differential equation above. In the time domain, this solution will be [8]

$$y(t) = e^{-\frac{b}{2m}t} \cdot \left[A_1 \cdot e^{\sqrt{\frac{b^2}{4m^2}-\omega_0^2}\cdot t} + A_2 \cdot e^{-\sqrt{\frac{b^2}{4m^2}-\omega_0^2}\cdot t} \right]$$

where $\omega_0 = \sqrt{\frac{k}{m}}$ is the frequency at which the system would oscillate when there would be no friction (b=0). If we define $\omega_1 = \sqrt{\omega_0^2 - \frac{b^2}{4m^2}}$ to be the frequency of the damped system, then we can rewrite the equation to:

$$y(t) = A \cdot e^{-\frac{b}{2m}t} \cdot \cos(\omega_1 \cdot t + \varphi)$$

A and φ are real integration constants in this equation, whose values depend on the initial state of the system. If we could determine the values of the spring constant k and the friction constant b we would be able to calculate the mass pushing on our weight bridge by determining the oscillation frequency and the damping ratio of the output signal. Figure 6.8 illustrates this.



a) *b)*

Figure 6.8: a) Measured Bridge Impulse Response
b) Simulated Impulse Response of Damped Mass-Spring System

Figure 6.8a shows the measured response to the knock on the bridge and figure 6.8b is the calculated curve of the above transfer function. The offset of the signals is due to the presence of an aluminium plate on the bridge. The damping and oscillation can be determined accurately from these plots. The procedure will be that we produce a damped oscillating signal with a calibrated weight. From the damping and the known value of m we can calculate the friction constant b. With these two variables the spring constant k can be determined from monitoring the oscillation frequency. Once the model's parameters are known we can calculate the mass by observing the output signals.

In practice, the cups will slide on the weight bridge with a certain speed and will roll off afterwards. This is comparable with an excitation of our damped mass-spring system with a pulse shaped like in figure 6.9.

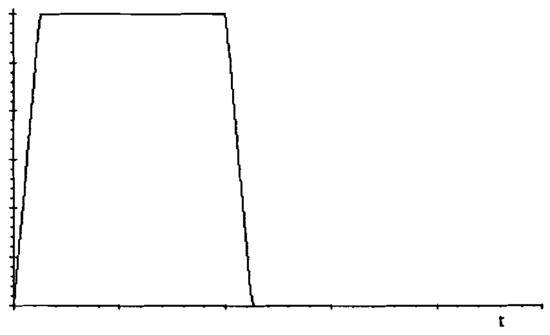


Figure 6.9: Excitation Signal of A Cup

The speed at which the pulse rises and falls depends on the speed of the cups. The faster the machine runs, the faster the cup is on the bridge and the faster the pulse reaches its final value. A simulation of the system's behaviour with an excitation like this produces the result of figure 6.10.

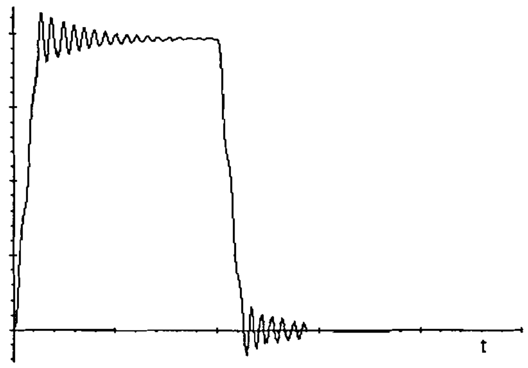


Figure 6.10: Simulated Cup Response of the Weight Bridge

As can be seen from this plot, the output is indeed that of a second order system. The signal overshoots its final value first, settles to that value and then rolls off quickly when the cup leaves the bridge, resulting in undershoot. Once again, if we can determine the frequency of the oscillations, either when the cup is on or off the bridge, and the damping of these signals, we would be able to calculate the mass of the fruit inside the cup very accurately.

Unfortunately, the bridge signals are not as 'neat' as in the simulation. Some signals coming out of the Wheatstone bridge are presented in figure 6.11. These signals are monitored with an oscilloscope on a fruit-grading machine. The machine is from the firm Agra and is running at a speed of 2.2 cups per second. The weights in the cup are calibrated.

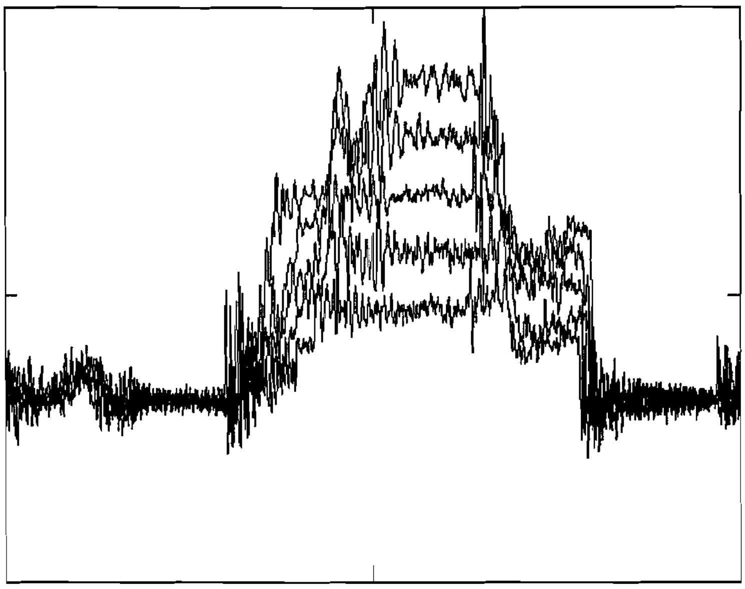


Figure 6.11: Bridge Output Signals

Figure 6.11 shows five different weights coming over the bridge, ranging from 100 grams to 500 grams in steps of 100 grams. The weight of the cup is the same for all the measurements. The signals suffer heavily from noise, due to machine vibrations. Furthermore, the cups are dragged over the bridge by chains, accelerated by chain wheels. This construction causes the cups to move in a slightly non-linear manner, resulting in the poor graphs of figure 6.11. Removing the noise out of the signal with a filter, does not lead to the desired plot of the simulated system in figure 6.10. Somehow, the machine or the non-linear movements cause the damped oscillations to vanish. Consequently, on an Agra machine, the mass of the fruit cannot be determined from the oscillation frequency and the damping. What we can do is filter the signal even more thoroughly to flatten the section of the signals where the cup is on the bridge and is oscillating to reach its final value. The resulting flattened curve is the weight of the cup with fruit in it. Figure 6.12 shows the filtered signals.

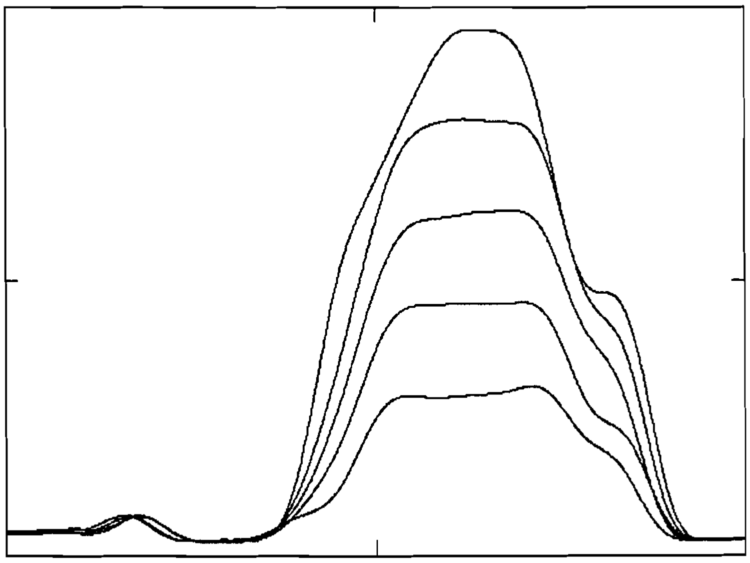


Figure 6.12: Filtered Bridge Output Signals

As can be seen from this figure, the output signals are quite flat, but are definitely not straight lines when the cup is on the bridge. If we take one value from this flat section, we might as well take a value that is a few grams to high or low. It is probably better to take the average of a small number of filtered values to represent the weight of that cup.

6.5 FIR Filter Algorithm

6.5.1 FIR Filter Design

A finite impulse response (FIR) filter is a discrete linear time-invariant system whose output is based on the weighted summation of a finite number of past inputs. FIR filters, unlike infinite impulse response (IIR) filters, are nonrecursive and require no feedback loops in their computation. This property allows simple analysis and implementation on microprocessors such as the ADSP-2171. A graphic representation of a FIR filter is shown in figure 6.13.

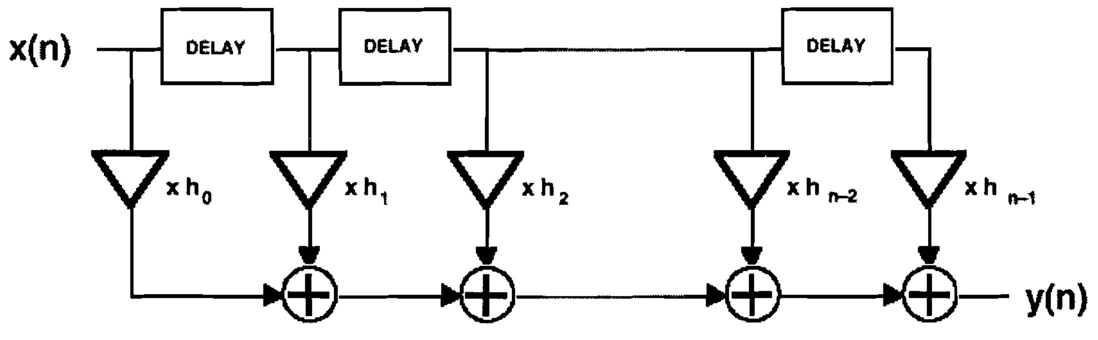


Figure 6.13: FIR filter

The realisation of a FIR filter can take many forms, although the most useful in practice is generally the transversal structure. A FIR transversal filter structure can be obtained directly from the equation for discrete-time convolution.

$$y(n) = \sum_{k=0}^{N-1} h_k(n) \cdot x(n-k)$$

In this equation, x(n) and y(n) represent the input to and output from the filter at time *n*. The output y(n) is formed as a weighted linear combination of the current and past input values of x, x(n–k). The weights, $h_k(n)$, are the transversal filter coefficients at time *n*. (For a nonadaptive filter, the coefficients do not change with *n*.) In the equation, x(n–k) represents the past value of the input signal "contained" in the (k+1)th tap of the transversal filter. For example, x(n), the present value of the input signal, would correspond to the first tap, while x(n–42) would correspond to the forty-third filter tap.

The FIR filter is designed using a Kaiser window. This window is defined to be

$$W[n] = \begin{cases} \dfrac{I_0\left(\beta\cdot\left(n-\dfrac{N-1}{2}\right)\right)}{I_0(\alpha)} & \text{for } 0 \le n < N \\ 0 & \text{otherwise} \end{cases}$$

where

$$\beta = \alpha\cdot\sqrt{1-\left(\frac{2n}{N-1}\right)^2}$$

and I_0 is the modified zero order Bessel function [9].

$$I_0(x) = 1 + \sum_{m=1}^{\infty}\left(\frac{\left(\frac{x}{2}\right)^m}{m!}\right)^2$$

By changing the parameter α we can change the suppression of the side-lobs of the filter response. Figure 6.14 shows the Kaiser window function for N=201 (201 taps), α=6, a sample frequency of 1kHz and a bandwidth of 1Hz. The resulting frequency response is drawn in figure 6.15.

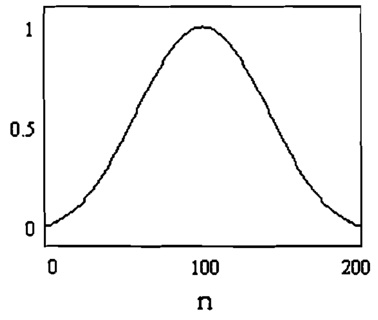


Figure 6.14: Kaiser Window

This filter is the same one that was used for the plots in figure 6.12. Note that both the DSP and the weights to be measured limit the number of taps and the bandwidth of the filter. The DSP has a finite time to calculate the entire FIR filter. The larger the number of taps, the better the filter will be but the more time the DSP needs to calculate the filter's response. Since the DSP is quite fast, the limiting factor in most cases will be the weights to be measured. The larger the number of taps, the longer it takes the filtered signal to reach its desired value. If the weight is too heavy, the filter won't reach the final value at all. This can be seen in figure 6.12, where the 'hills' get sharper when the weight increases. For even higher weights than 500 grams the filter's response would have been too slow.

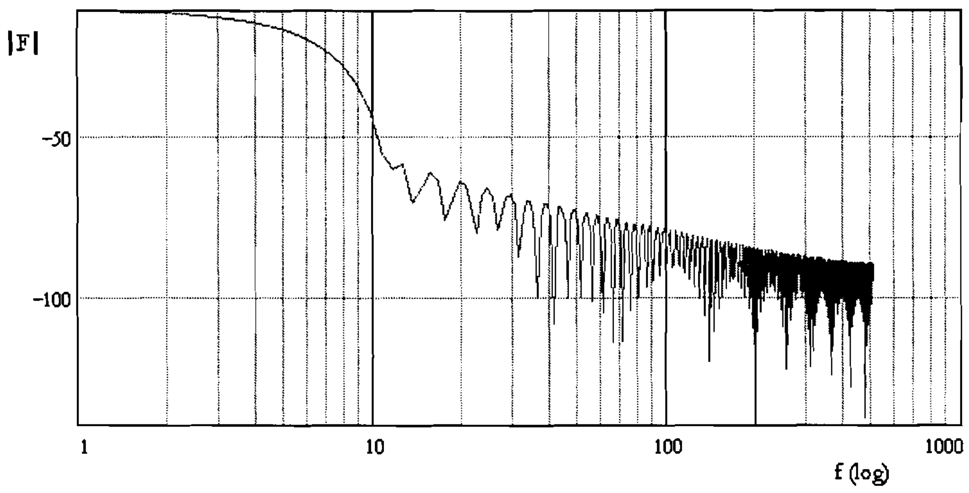


Figure 6.15: Filter Response

The results of measuring with the above filter, using 201 taps and averaging over 10 filtered values, are listed in table 6.3.

Table 6.3: Results With FIR Filter Implementation

Calibrated weight	Calculated weight in grams, 1.0 cups per second, delay = 650 ms			Calculated weight in grams, 2.2 cups per second, delay = 410 ms		
	1	*2*	*3*	*1*	*2*	*3*
100	100	100	100	99	100	99
200	204	204	205	202	202	201
300	301	301	304	298	302	302
400	401	401	402	399	401	401
500	499	501	500	496	498	496

It is clear that due to the poor mechanical behaviour of the Agra machine a precision of 1 gram cannot be established, even with a large FIR filter. An accuracy of ±2 grams is more realistic for these machines. Note that for 500 grams at a speed of 2.2 cups per second, the FIR filter is too slow to calculate the weight. Lowering the number of taps might increase the reliability of these measurements at the cost of reduced filter performance and thus measurement accuracy for lower weights. In general, a trade-off must be made between high accuracy for low weights (long FIR filter possible) or lower accuracy for a larger range of weights (shorter FIR filter necessary).

6.5.2 FIR Filter Software Implementation

The software for this FIR filter uses a runlevel scheme with one additional runlevel, the *Filtering* runlevel, as shown in figure 6.16.

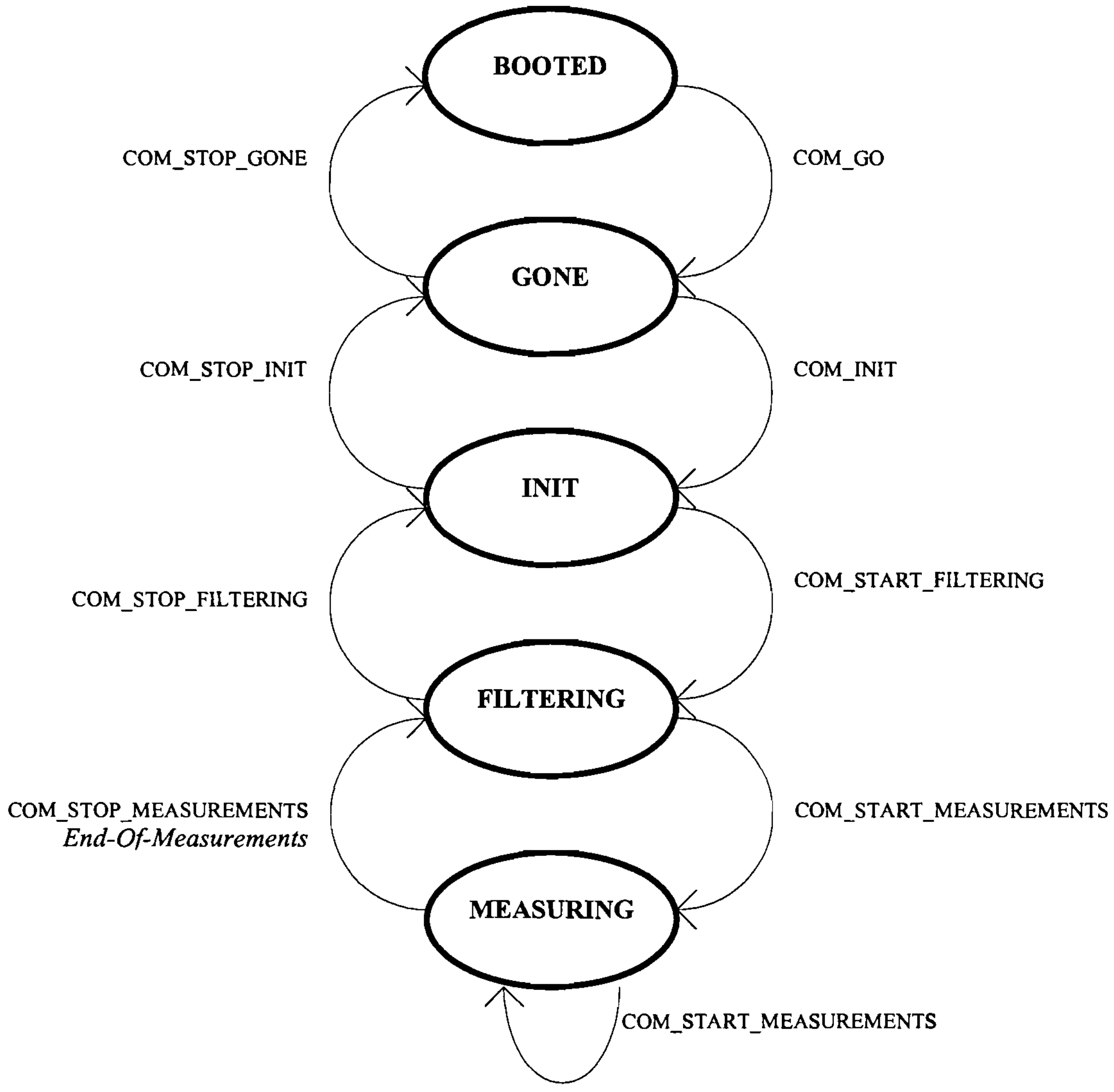


Figure 6.16: Filter Runlevels

In this new runlevel, the DSP is filtering all incoming data with the downloaded FIR filter. Note that this does not imply any measurement action, but only the filtering of the signals. The actual measuring is done in the *Measuring* runlevel. This runlevel takes the average of a given number of samples and takes this to be the weight of the measured cup. The additional commands that are used for the FIR filter implementation are listed in table 6.4. A complete command overview can be found in Appendix F.

Table 6.4: Additional FIR Filter Command Overview

Runlevel	Command	Description
Init	COM_FILTER_LEN	Set the desired length of the FIR-filter.
	COM_SET_SAMPLES	Set the number of samples from which the average will be calculated.
	COM_SET_FILTER_CONST	Set the filter constants. The filter length determines the number of constants.
	COM_START_FILTERING	Start filtering on all channels, even if certain channels are not used.
Filtering	COM_GET_FILTERED_CURVE	Returns a curve of filtered data. Parameters are the channel number, the number of points and the sample frequency.
	COM_STOP_FILTERING	Stop filtering and return to a lower runlevel.
	COM_START_MEASUREMENTS	Starts measuring on the selected channels.

The measurement jobs use some extra variables to keep track of the state of the data while it passes through the FIR filter.

```
typedef struct jobst
{
 uint16 active;          /* 0=inactive job, 1=active job                     */
 uint16 cup_number;      /* number of the measured cup                       */
 uint16 timer[8];        /* counts down until delay time has passed          */
 uint16 counter[8];      /* counts down until data is rippled through the filter  */
 int32  weight_sum[8];   /* sum of ten weighted values of this cup           */
 uint16 weight_count[8]; /* number of weights for average                    */
} jobt;
```

The `counter` array is initially set to the filter length and is decremented each time new data for a channel has arrived. Once it reaches zero, the data that is coming out of the FIR filter is the data that was to be measured. The `weight_sum` array contains the sum of the filtered values for each channel to be able to calculate the averages. The `weight_count` is initially filled with the number of samples for this average and is decremented each time new data arrives for a channel. Once it reaches zero, the average of `weight_sum` can be calculated and the measurement for that channel is finished.

The complete source of the FIR filter function can be found in Appendix D. This FIR filter algorithm requires 86+N cycles plus some interrupt overhead, where N is the number of taps. When running at a rate of 24Mhz, a 101-taps filter will be calculated within 7.8 microseconds. Figure 6.17 shows the percentage of time that is used for the FIR filter algorithm at full load (four measurement jobs).

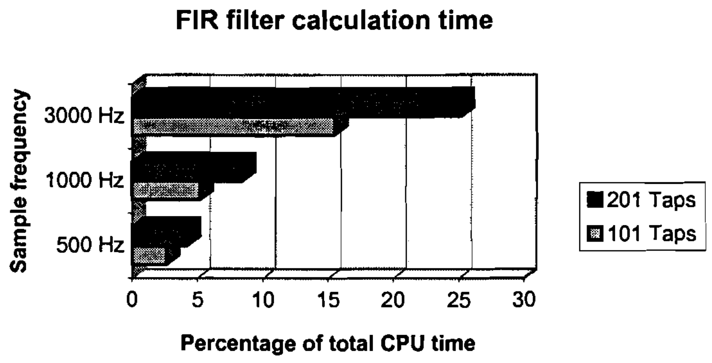


Figure 6.17: FIR Filter Calculation Time

Although the A/D-converter can sample up to 100kHz, the practical limit of the sample frequency is 24kHz. In this case, every channel is sampled at a rate of 3kHz and the DSP can serve these interrupts just in time when using an instruction rate of 24Mhz (24 MIPS).

7. HOST INTERFACE PORT

7.1 Overview

The ADSP-2171 digital signal processor has a host interface port (HIP). The HIP is a parallel I/O port that allows the processor to be used as a memory-mapped peripheral of a host processor. Examples of host processors include the Intel 8051, Motorola 68000 family and even older ADSP-21xx processors [10].

The host interface port can be thought of as an area of dual-ported memory or mailbox registers that allow communication between the host and the processor core of the ADSP-2171. The host addresses the HIP as a segment of 8- or 16-bit words of memory. To the processor core, the HIP is a group of eight data-memory-mapped registers.

The operation speed of the HIP is similar to that of the processor data bus. A read or write operation can occur within a single instruction cycle. Because the HIP is normally connected to devices that are much slower, the host processor usually limits the data transfer rate.

The host interface port is completely asynchronous to the rest of the ADSP-2171's operations. The host can write data to or read data from the HIP while the ADSP-2171 is operating at full speed. The HIP can be configured for operation on an 8-bit or 16-bit data bus and for either a multiplexed address/data bus or separate address and data buses.

7.2 HIP Functional Description

The HIP consists of three functional blocks, shown in figure 7.1: a host control interface block (HCI), a block of six data registers (HDR5-0) and a block of two status registers (HSR7-6). The HIP also includes an associated HMASK register for masking interrupts generated by the HIP. The HCI provides the control for reading and writing the host registers. The two status registers provide status information to both the host and the ADSP-2171 core.

The HIP data registers HDR5-0 are memory-mapped into internal data memory at locations 0x3FE0 (HDR0) to 0x3FE5 (HDR5). These registers can be thought of as a block of dual-ported memory. None of the HDRs are dedicated to either direction; they can be read or written by either the host or the ADSP-2171. When the host reads an HDR register, a maskable HIP read interrupt is generated. When the host writes an HDR, a maskable HIP write interrupt is generated.

The read/write status of the HDRs is also stored in the HSR registers. These status registers can be used to poll HDR status. Thus, data transfers through the HIP can be managed by using either interrupts or a polling scheme, described later in this chapter.

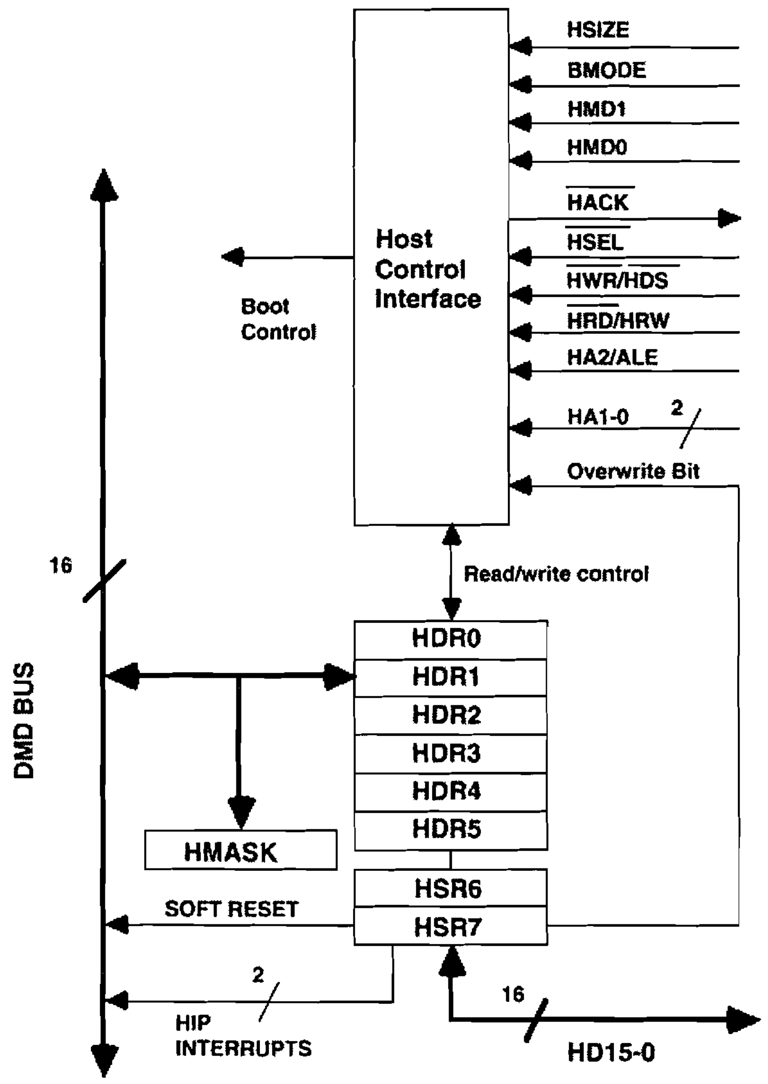


Figure 7.1: HIP Block Diagram

The HSR registers are shown in figure 7.2. Status information in HSR6 and HSR7 shows which HDRs have been written. The lower byte of HSR6 shows which HDRs have been written by the host computer. The upper byte of the HSR6 shows which HDRs have been written by the ADSP-2171. When an HDR register is read, the corresponding HSR bit is cleared.

The lower six bits of HSR7 are copied from the upper byte of HSR6 so that 8-bit hosts can read both sets of status. Bits 7 and 6 of HSR7 control the overwrite mode and software reset, respectively; these functions are described later in this chapter. The upper byte of HSR7 is reserved. All reserved bits and the software reset bit read as zeros. The overwrite-bit is the only bit in the HSRs that can be both written and read. At reset, all HSR bits are zeros except for the overwrite-bit, which is a one.

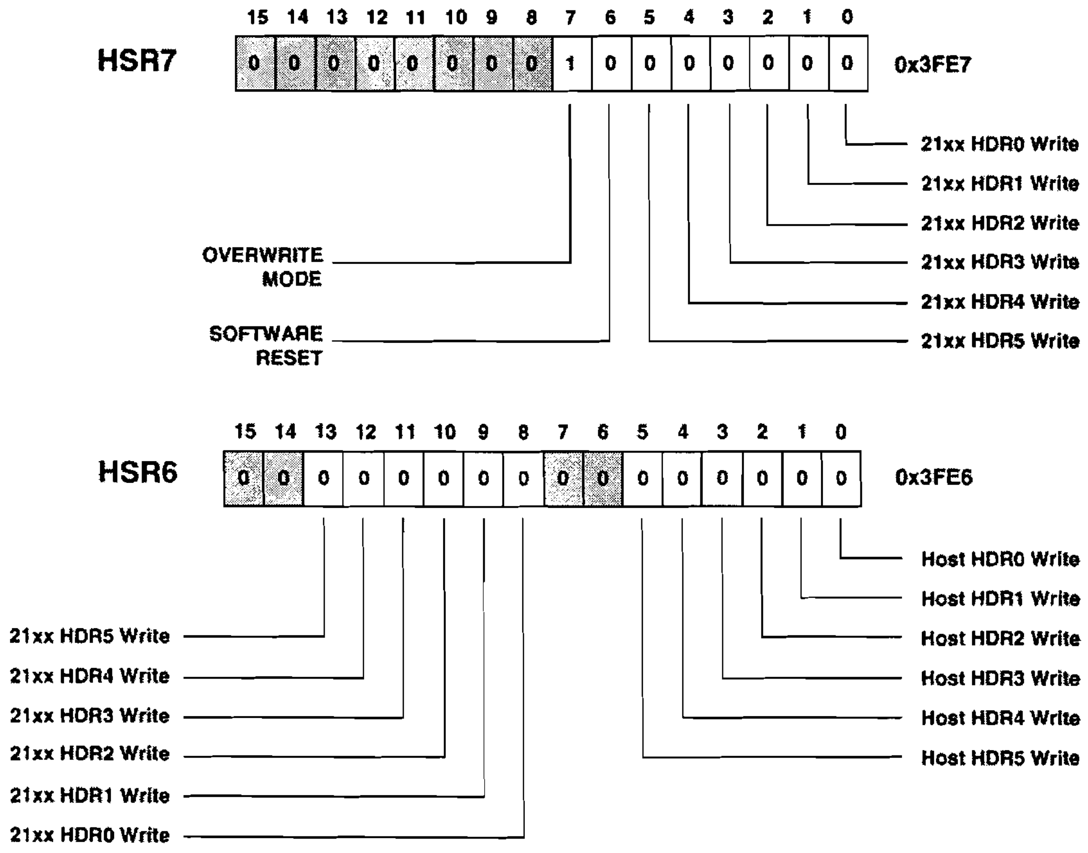


Figure 7.2: HIP Status Registers

7.3 HIP Operation

The ADSP-2171 core can place a data value into one of the HDRs for retrieval by the host computer. Similarly, the host computer can place a data value into one of the HDRs for retrieval by the ADSP-2171. To the host computer, the HDRs function as a section of memory. To the ADSP-2171, the HDRs are memory-mapped registers, part of the internal data memory space.

Because the HIP typically communicates with a host computer that has both a slower instruction rate and a multicycle bus cycle, the host computer is usually the limiting factor in the speed of HIP transfers. During a transfer, the ADSP-2171 executes instructions normally, independent of HIP operation. This is true even during a multicycle transfer from the host.

For host computers that require handshaking, the ADSP-2171 returns HACK in the same cycle as the host access, except in overwrite mode. In overwrite mode, the ADSP-2171 can extend a host access by not asserting the HACK acknowledge until the cycle is complete. The user can enable and disable overwrite-mode by setting and clearing a bit in HSR7. Overwrite mode is described in more detail later in this chapter.

The HDRs are not initialised during either hardware or software reset. The host can write information to the HDRs before a reset, and the ADSP-2171 can read this information after the reset is finished. During reset, however, HIP transfers cannot occur; the HACK pin is deasserted and the data pins are tristated.

Because a host computer that requires handshaking must wait for an acknowledgement from the ADSP-2171, it is possible to cause such a host to hang. If, when the host has initiated a transfer, but has not yet received an acknowledgement, the ADSP-2171 is reset, then the acknowledgement can not be generated, thus causing the host to wait indefinitely.

There is no hardware in the HIP to prevent the host from writing a register that the ADSP-2171 core is reading (or vice versa). If the host and the ADSP-2171 try to write the same register at the same time, the host takes precedence. Simultaneous writes should be avoided, however: since the ADSP-2171 and the host operate asynchronously, simultaneous writes can cause unpredictable results.

7.3.1 Polled Operation

Polling is one method of transferring data between the host and the ADSP-2171. Every time the host writes to an HDR a bit is automatically set in the lower byte of HSR6. This bit remains set until the ADSP-2171 reads the HDR. Similarly, when the ADSP-2171 writes to an HDR, a bit in the upper byte of HSR6 (and the lower byte of HSR7) is set. This bit is cleared automatically when the host reads the HDR.

For example, the ADSP-2171 can wait in a loop reading an HSR bit to see if the host has written new data. When the ADSP-2171 sees that the bit is set, it conditionally jumps out of the loop, processes the new data, then returns to the loop. When transferring data to the host, the ADSP-2171 waits for the host to read the last data written so that new data can be transferred. The host polls the HSR bits to see when the new data is available.

7.3.2 Interrupt-Driven Operation

Using an interrupt-driven protocol frees the host and the ADSP-2171 from polling the HSR(s) to see when data is ready to be read. For interrupt-driven transfers to the ADSP-2171, the host writes data into an HDR, and the HIP automatically generates an internal interrupt. The interrupt is serviced like any other interrupt.

For transfers to the host, the ADSP-2171 writes data to an HDR, then sets a flag output, which is connected to a host interrupt input, to signal the host that new data is ready to be transferred. If the ADSP-2171 passes data to the host through only one HDR, then that HDR can be read directly by the host when it receives the interrupt. If more than one HDR is used to pass data, then the host must read the appropriate HSR(s) to determine which HDR was written by the ADSP-2171.

7.3.3 HDR Overwrite Mode

In most cases, the ADSP-2171 reads host data sent through the HIP faster than the host can send them. However, if the host is sufficiently fast, if the ADSP-2171 is busy, or if the ADSP-2171 is driven by a slow clock, there may be a delay in servicing a host write interrupt. If the host computer uses a handshaking protocol requiring the ADSP-2171 to assert HACK to complete a host transfer, the ADSP-2171 can optionally hold off the next host write until it has processed the current one.

If the HDR overwrite bit (bit 7 in HSR7) is cleared, and if the host tries to write to a register before it has been read by the ADSP-2171, HACK is not asserted until the ADSP-2171 has read the previously written data. The host processor must wait for HACK to be asserted. As described earlier, however, there is a delay from when the host writes data to when the status is synchronised to the ADSP-2171. During this interval, it is possible for the host to write an HDR a second time even when the overwrite-bit is cleared.

If the HDR overwrite bit is set, the previous value in the HDR is overwritten and HACK is returned immediately. If the ADSP-2171 is reading the register that is being overwritten, the result is unpredictable. After reset, the HDR overwrite bit is set. If the host does not require an acknowledge (HACK is not used), the HDR overwrite bit should always be set, because there is no way for the ADSP-2171 to prevent overwrite.

7.3.4 Software Reset

Writing a 1 to bit 6 of HSR7 causes software reset of the ADSP-2171. If the ADSP-2171 writes the software-reset bit, the reset happens immediately. Otherwise, the reset happens as soon as the write is synchronised to the ADSP-2171 system clock. The internal software reset signal is held for five ADSP-2171 clock cycles and then released.

7.4 HIP Interrupts

HIP interrupts can be masked using either the IMASK register or the HMASK register. Bits in the IMASK register enable or disable all HIP read interrupts or all HIP write-interrupts. The HMASK register, on the other hand, has bits for masking the generation of read- and write-interrupts for individual HDRs, see figure 7.3. In order for a read or write of an HDR to cause an interrupt, the HIP read or write interrupt must be enabled in IMASK, and the read or write to the particular HDR must be enabled in HMASK. HMASK is mapped to memory location 0x3FE8.

A host write-interrupt is generated whenever the host completes a write to an HDR. A host read interrupt is generated when an HDR is ready to receive data from the ADSP-2171—this occurs when the host has read the previous data, and also after reset, before the ADSP-2171 has written any data to the HDR. HMASK, however masks all HIP interrupts at reset. The read interrupt allows the ADSP-2171 to transfer data to the host at a high rate without tying up the ADSP-2171 with polling overhead. HMASK allows reads and writes of some HDRs to not generate interrupts. For example, a system might use HDR2 and HDR1 for data values and HDR0 for a command value. Host write-interrupts from HDR2 and HDR1 would be masked off, but the write interrupt from HDR0 would be unmasked, so that when the host wrote a command value, the ADSP-2171 would process the command. In this way, the overhead of servicing interrupts when the host writes data values is avoided.

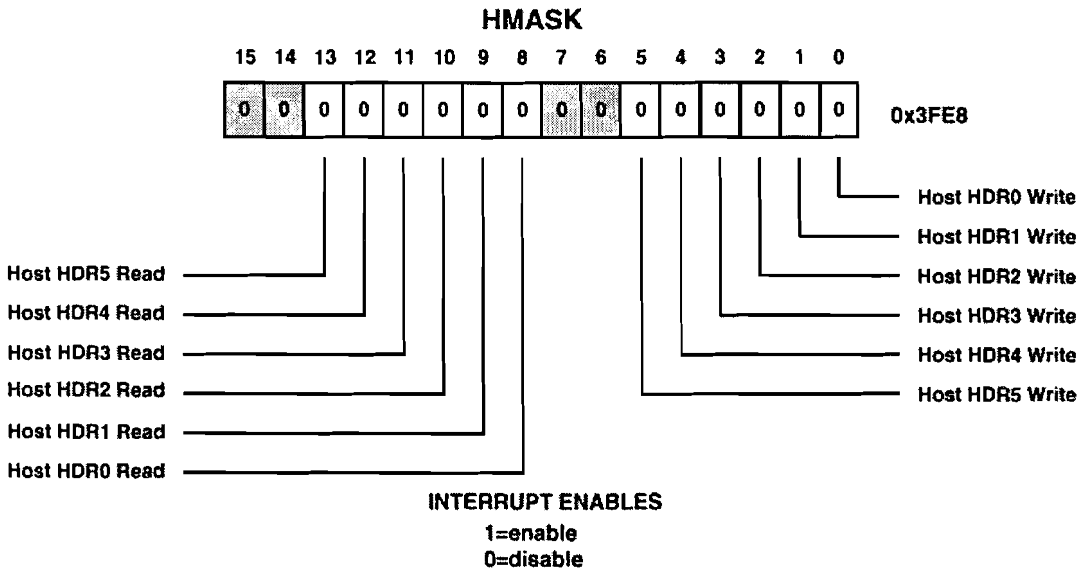


Figure 7.3: HMASK Register

The HMASK register is organised in the same way as HSR6; the mask bit is in the same location as the status bit for the corresponding register. The lower byte of HMASK masks host write-interrupts and the upper byte masks host read interrupts. The bits are all positive sense (0=masked, 1=enabled). HMASK is mapped to the internal data memory space at location 0x3FE8. At reset, the HMASK register is all zeros, which means that all HIP interrupts are masked.

HIP read and write-interrupts are not cleared by servicing such an interrupt. Reading the HDR clears a write interrupt, and writing the HDR clears a read interrupt. The logical combination of all read- and write-interrupt requests generates a HIP interrupt. Pending interrupt requests remain until all HIP interrupts are cleared by either reading or writing the appropriate HIP data register. If the ADSP-2171 is reading registers that the host might be writing, it is not certain that an interrupt will be generated. To ensure that all host writes generate interrupts, you must make sure that the ADSP-2171 is not reading the HDRs that the host is writing. While servicing the interrupt, the status register can be read to determine which operation generated the interrupt and whether multiple interrupt requests need to be serviced.

HIP interrupts cannot be forced or cleared by software, as other interrupts can. The HIP write-interrupt vector is location 0x0008. The HIP read interrupt vector is location 0x000C.

7.5 Booting through the HIP

The entire internal program RAM of the ADSP-2171, or any portion of it, can be loaded using a boot sequence. Upon hardware or software reset, the boot sequence occurs if the MMAP pin is 0. If the MMAP pin is 1, the boot sequence does not occur.

The ADSP-2171 can boot in either of two ways: from external memory (usually EPROM), through the boot memory interface, or from a host processor, through the HIP. The BMODE pin selects which type of booting occurs. When the BMODE=1, booting occurs through the HIP.

Booting through the HIP occurs in the following sequence:

1. After reset, the host writes the length of the boot sequence to HDR3.
2. The host waits at least two ADSP-2171 processor cycles.
3. Starting with the instruction which is to be loaded into the highest address of internal program memory, the host writes an instruction into HDR0, HDR2 and HDR1 (in that order), one byte each. The upper byte goes into HDR0, the lower byte goes into HDR2 and the middle byte goes into HDR1.
4. The address of the instruction is decremented, and Step 3 is repeated. This continues until the last instruction has been loaded into the HIP.

The ADSP-2171 reads the length of the boot load first, then bytes are loaded from the highest address downwards. This results in shorter booting times for shorter loads. The number of instructions booted must be a multiple of eight. The boot length value is given as:

$$\text{length} = \frac{\text{number of 24-bit program memory words}}{8} - 1$$

That is, a length of 0 causes the HIP to load eight 24-bit words. In most cases, no handshaking is necessary, and the host can transfer data at the maximum rate it is capable of. If the host operates faster than the ADSP-2171, wait states or NOPs must be added to the host cycle to slow it down to one write every ADSP-2171 clock cycle.

The following example shows the data that a host would write to the HIP for a 1000-instruction boot:

Data	*Location*
Page Length (124 decimal)	HDR3
Upper Byte of Instruction at 999	HDR0
Lower Byte of Instruction at 999	HDR2

Middle Byte of Instruction at 999	HDR1
Upper Byte of Instruction at 998	HDR0
Lower Byte of Instruction at 998	HDR2
Middle Byte of Instruction at 998	HDR1
Upper Byte of Instruction at 997	HDR0
Lower Byte of Instruction at 997	HDR2
Middle Byte of Instruction at 997	HDR1
●	
●	
●	
Upper Byte of Instruction at 0	HDR0
Lower Byte of Instruction at 0	HDR2
Middle Byte of Instruction at 0	HDR1

A 16-bit host boots the ADSP-2171 at the same rate as an 8-bit host. Either type of host must write the same data to the same the HDRs in the same sequence (HDR0, HDR2, and HDR1). If a 16-bit host writes 16-bit data, the upper byte of the data must be 0x00. The following example, loading the instruction 0xABCDEF, illustrates this:

	8-Bit Host	*16-Bit Host*
1st Write (to HDR0)	0xAB	0x00AB
2nd Write (to HDR2)	0xEF	0x00EF
3rd Write (to HDR1)	0xCD	0x00CD

7.6 Used HIP Implementation

In chapter 9 we will discuss the host processor and its operation. An 8-bit host processor will be chosen for our fruit grading system. This means that the HIP must be configured to operate in 8-bit mode as well. The host processor uses a multiplexed address/data bus, so the HIP is also configured to use this kind of architecture.

Since the ADSP-2171's main task is filtering and analysing incoming signals, polling the HIP to see whether or not the host has written new data to it, is not efficient. The software is set up to use interrupts each time the host writes or reads from a HIP register. When the ADSP-2171 writes a value into one of its HIP registers, the host is interrupted by a flag output of the ADSP-2171. This allows the host processor to operate in an interrupt-based manner as well.

To prevent simultaneous writes by the host and the DSP, the six data registers are divided into two parts of three registers each as shown in figure 7.4. The lower three registers are used for messages from the host to the DSP and the upper three registers are used for messages from the DSP to the host. In this way, no collisions can occur during normal operation.

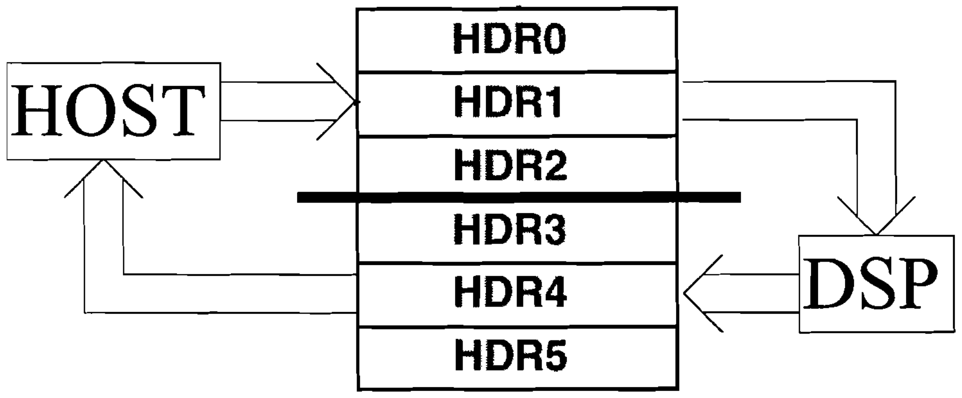


Figure 7.4: HIP Communication

The protocol that is used to establish a secure communication channel is straightforward. HDR0 is used as the command register in which the host processor puts the command that must be executed. The other two registers, HDR1 and HDR2, are used to transfer data values if present. On the other side, the same layout is used. HDR3 is the DSP status register in which the DSP puts its return messages. The other two registers, HDR4 and HDR5, are used to transfer data values.

Each time the host writes a new command in HDR0, the DSP will generate an internal interrupt. The DSP is configured in such a way that only HDR0 will produce a HIP write-interrupt. The DSP can answer in two ways: an acknowledgement by putting HIP_ACK in HDR3, or an error message. If the DSP sends an acknowledgement, the host can proceed with sending data values if they are required for that specific command. Data values are set in HDR1 and HDR2 and after setting them up, the host puts a NEW_DATA command in HDR0. Each data transmission is acknowledged by the DSP with a HIP_ACK return message. Transfers from the DSP to the host operate in the same manner. The host acknowledges each transfer with a HIP_ACK in HDR0. Figures 7.5a, 7.5b and 7.5c illustrate what happens during normal transmission and transmissions in which an error occurs.

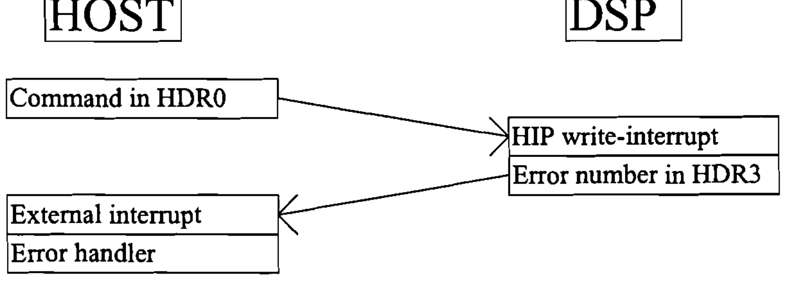


Figure 7.5a: Transfer from host to DSP with error.

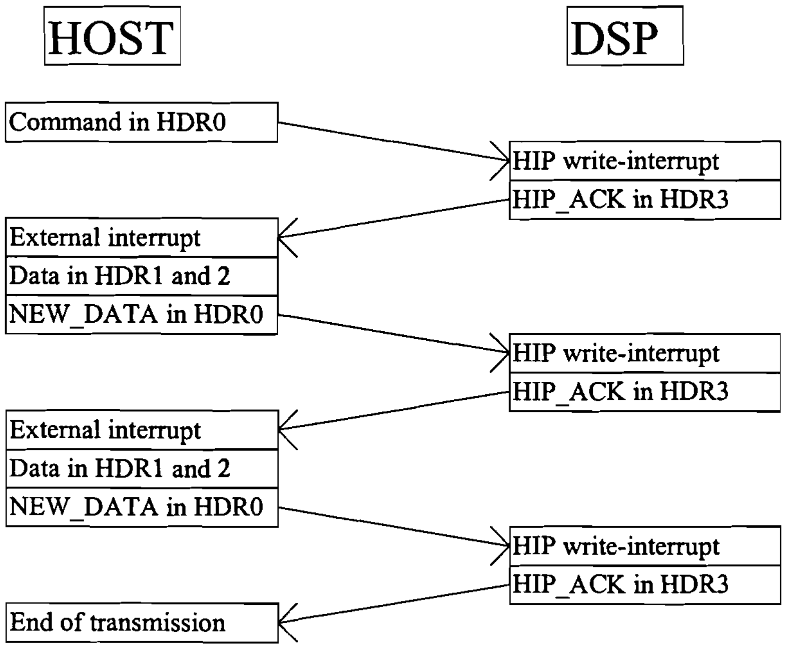


Figure 7.5b: Transfer from host to DSP, no errors.

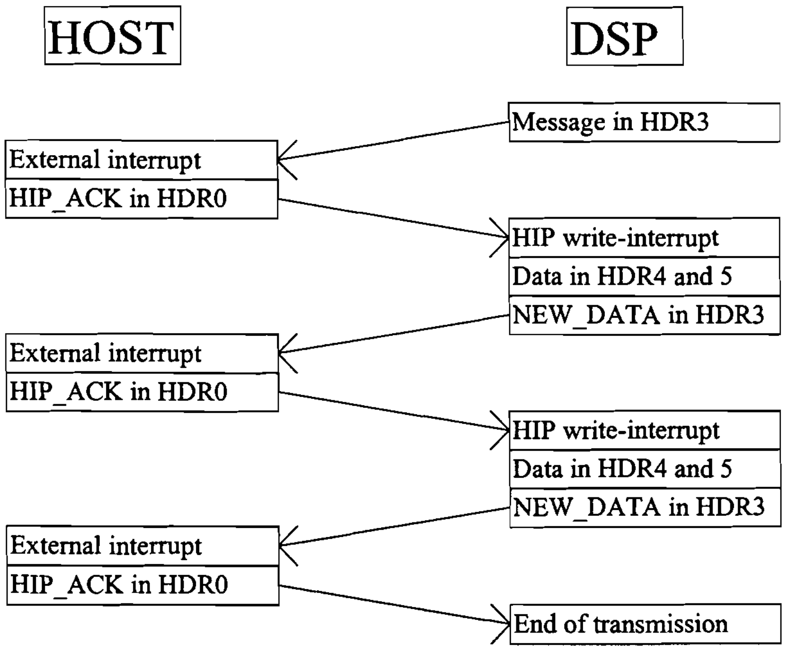


Figure 7.5c: Transfer from DSP to host, no errors.

The external interrupt from the host is generated by one of the flag output pins of the ADSP-2171. The external interrupt is cleared whenever the host reads a data value from HDR3, HDR4 or HDR5. This read action from the host causes the DSP to generate a HIP read interrupt. The HIP read interrupt routine clears the flag output pin and clears the internal interrupt of the DSP by reading the contents of HDR3, 4 and 5 and writing them to these registers again. Since writing to the HIP-registers is the only way of clearing the HIP read interrupt, this method is the only possibility of clearing that interrupt without destroying the contents of the HIP registers.

8. PROFIBUS INTERFACE

8.1 Profibus ASIC

Several ASICs (Application Specific Integrated Circuits) are available to take care of the lowest Profibus layer. These dedicated chips drastically simplify the implementation of the Profibus protocol. For our application, we need an ASIC that is capable of communicating at speeds up to 12 Mbit/s. Furthermore, we wish to be able to implement Profibus-DP, both cyclic and a-cyclic, in future versions of the fruit grading system. Siemens AG offers the SPC-series (Siemens Profibus Controller) to be used as Profibus interface. Both the SPC3 and the SPC4 are suited for our application, but the software provided by Siemens does only include cyclic Profibus-DP, not the a-cyclic extensions. However, another company does support a-cyclic Profibus-DP communications in software for the SPC4, so this ASIC will be used in the weight controller and other Profibus slave devices.

The main features of the SPC4 are:
- complete layer 1 fulfilling
- filtering of erroneous telegrams
- bit rates from 9.6 kbit/s up to 12 Mbit/s
- interface to 8-bit microprocessors using multiplexed or separate data/address bus
- communication with the microprocessor through 1 kilobytes of Dual Ported Ram
- indication buffer to indicate which SAP has received a service request

Additional hardware is required to attach the SPC4 to the Profibus cable. A standard interface connection described in all Profibus specifications is sufficient to perform this task. The Profibus signals are not directly fed to the weight controller but are directed through opto-couplers. If by accident hazardous voltages appear on the Profibus cable, only the opto-couplers will be damaged and not the more expensive SPC4.

8.2 SPC4 Software

The software for the SPC4 is supplied by TMG i-tec GmbH. It consists of several modules in the C language, taking care of the FDL-layer (layer 2) and all Profibus-DP and DPE functions. The user can rapidly build its own application by initialising some variables and calling the proper functions in the various modules.

Since the first version of the weight controller will not use any DP or DPE functions but accesses the FDL-layer directly, we only need the basic FDL-functions. These functions are not described in the manuals, so a brief overview is presented here:
- `void sap_activate(byte sap, sap_verwaltung *sap_config);`
  used to activate a service access point (SAP)
- `void sap_xaccess(byte sap_nr, byte access, byte sap);`
  used to change the access to a SAP

- `puffer *get_sap_puffer(byte sap);`
  returns a pointer to a buffer (German: *puffer*) in which the user can fill in the data that has to go to the master computer. The user has to fill in the following items:

♦ `puffer.modus`	Transmit mode (MULTIPLE or SINGLE) and priority (LOW or HIGH).
♦ `puffer.laenge`	Length of the data in lli_pdu.
♦ `puffer.lli_pdu[245]`	The user data, depends on the application.

- `fdl_reply_update(puffer *point);`
  sends the buffer, filled by the user, to the master computer and frees the buffer space for future use. The buffer has to be requested first with the `get_sap_puffer` function.

The `puffer` type contains the following items:

```
typedef struct pufferinhalt
{
  void *next;               linked list of empty buffers
  byte reservedlli;         for DPE only
  byte fdl_kr;              not used
  byte modus;               transmit mode (MULTIPLE, SINGLE) and
                            priority (LOW, HIGH)
  byte laenge;              length of data in lli_pdu
  byte pa;                  remote address
  byte osap;                local SAP
  byte psap;                remote SAP
  byte lli_pdu[245];        user data
} puffer;
```

The `sap_verwaltung` type for activation of a SAP is defined as:

```
typedef struct
{
  byte access;                  remote FDL address access or ALL
  byte access_sap;              remote FDL sap access or ALL
  byte akt_dienste;             activated services
  byte max_fdl_laenge_s;        maximum length of send buffer
  byte max_fdl_laenge_r;        maximum length of receive buffer
  queue leer_schlange;          queue of empty buffers
  byte ChipSegment;             page in SPC4
  pub_list *publisher;          not used
} sap_verwaltung;
```

The user application should check repeatedly whether or not an indication for the used SAPs has been received. If the indication is received, the SPC4 has already sent the previously filled buffer. The user can now ask for a new buffer with `get_sap_puffer`, fill it with the requested data and put it in the SPC4 with the `fdl_reply_update` function. Next time the master polls the device with this SAP, the reply is transmitted.

9. HOST OPERATION

9.1 Host Processor

The host processor has to communicate with both the DSP and the SPC4. Basically, it doesn't do much more than passing messages between those two components. One could wonder if the DSP would not be able to communicate with the SPC4 directly. Didn't figure 6.17 show us that the DSP uses less than 6% of its CPU power under normal operating conditions? Indeed, the CPU power is sufficient to perform this extra task, but the ADSP-2171 has a 16-bit architecture and cannot write bytes to a memory location. This means that the Dual Ported Ram of the SPC4 cannot be accessed without special tricks. It is cheaper and easier to use an extra micro controller instead.

The I/O-card uses an 80C32 micro controller to communicate with the SPC4 and to switch the relays. Since we don't like to master several development environments, we will choose the 80C32 for the weight controller as well. It is an 8-bit micro controller which main features are (see also figure 9.1):

- Fully compatible to standard 8051 micro controller
- Versions for 12/24/40 MHz operating frequency
- 256 × 8 RAM
- Four 8-bit ports
- Three 16-bit timers / counters (timer 2 with up/down counter feature)
- USART
- Six interrupt sources, two priority levels
- Power saving modes

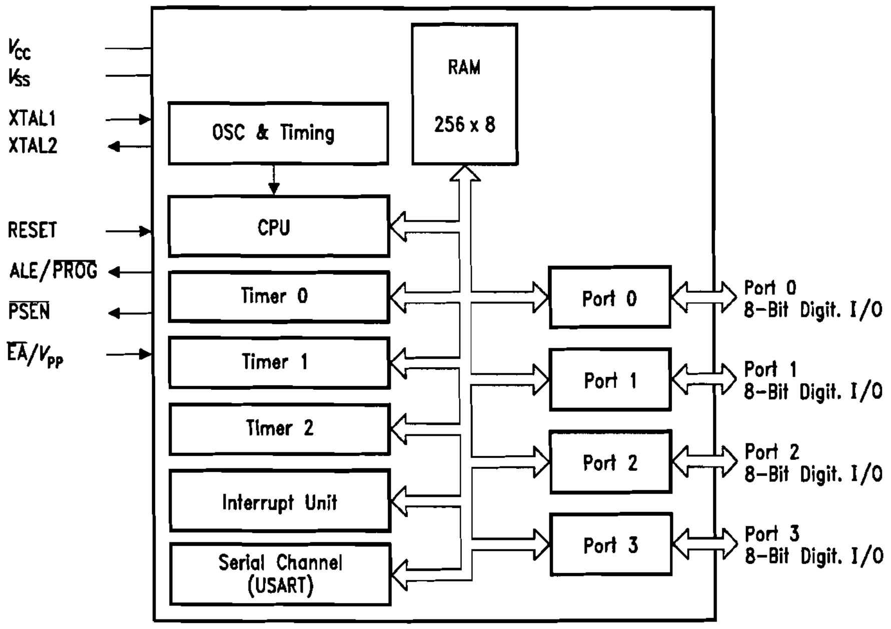


Figure 9.1: 80C32 Diagram

With the $\overline{EA}$ input connected to ground, all program fetches are directed to external memory. Up to 64 kilobytes can be addressed in this way. Apart from this external memory, 256 bytes of internal memory are present as well as 128 bytes in a special function register area.

9.2 Memory Interface

The host processor will need some non-volatile memory that can store its program. Apart from that, ordinary static RAM (SRAM) is necessary for storing variables. We could use an EEPROM as program memory, but this would be inconvenient with respect to software updates. For a fruit grading system abroad this would mean that a new EEPROM has to be shipped every time the software is changed. Flash memory has the advantage that the host processor itself can program it. After power on, the host will transfer its program from flash memory to SRAM and run from SRAM afterwards. If the master computer sends new software, the host processor will store it in flash memory, from which it can boot next time or right away. This results in a very flexible upgrade service, since the actual program code can be send to the master station over the phone line using two modems.

In addition, the DSP software can be stored in the same flash memory. The host processor can send it through the HIP in the boot sequence as described in paragraph 7.5. No extra hardware is needed to boot the DSP in this way.

The host processor can perform all of the above actions if it is able to choose from one of three memory configurations, shown in figure 9.2. The left side in each figure is program memory, the right side is data memory.

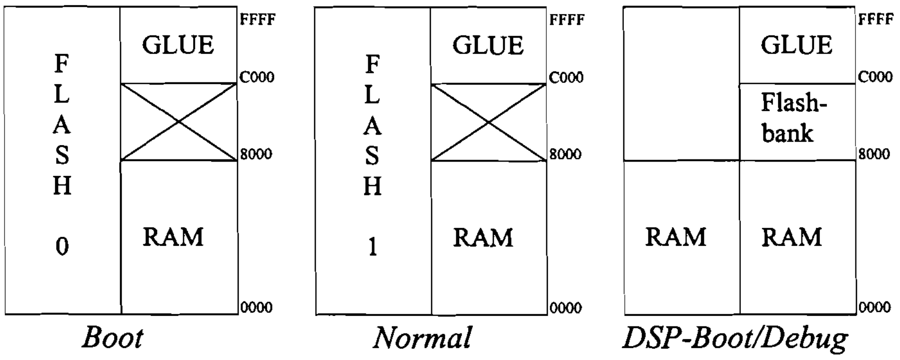


Figure 9.2: Host Processor Memory Configurations

After power-on the host will start in the *boot*-configuration, in which it will run its program from flash-memory. The flash memory is 128kB and only the lower 64kB will be used in this state. The processor will start running from address 0 and this is where its "survival code" is located. This piece of code allows the host to download new software over the Profibus. In this way, the system will always run, even if by accident the rest of the host software is erased.

The upper 64kB of the flash memory is used in the *normal*-configuration. This allows the host to run a different program by simply switching the memory configuration. The lower 64kB of the flash-memory can for example be filled with the "survival-code" and the DSP software, while the actual weight functions are located in the upper 64kB of the flash-memory.

The third memory mode is the *DSP-Boot/Debug* mode. In this mode, the program and data memory are the same 32kB of RAM and a 16kB bank of flash-memory can be mapped in data memory. The DSP software in flash-memory can therefore be read entirely and sent through the HIP to the ADSP-2171. Furthermore, the entire flash-memory can be written to update the host software. This mode is also used for debug purposes, since the debugger uses some self-modifying-code tricks that cannot be employed with flash-memory. Note that in this mode the program must first be copied from flash-memory to RAM before switching.

The memory mode can be written to and read from a special register, stored in the Glue logic. The flash bank is selected with a second write-only register in the Glue (see paragraph 9.3).

Bus arbitration is necessary, since only one multiplexed data/address bus is used to address the flash memory, the SRAM, the SPC4 and the HIP. Furthermore, the different memory modes imply that program instructions are fetched from both RAM and flash-memory. The selection of the appropriate chips is done by the Glue logic.

9.3 Glue Logic

The Glue logic is the piece of hardware that makes all the other hardware work in harmony. It "glues" together all the individual memory components, processors and other peripherals. For the weight controller, its functions are:

- controlling memory addressing
- switching the multiplexer
- starting and controlling A/D-conversions
- supplying a 12 MHz clock signal for the DSP

These tasks can be accomplished by using an FPGA and programming it with the VHDL-language. For our purpose, the Quick-Logic QL8x12B, QL12x16B and the newer QL2003 can be used. These FPGAs are pin-compatible and differ only in size and price.

The 12 MHz clock signal for the DSP is simply derived from the 24 MHz clock signal coming out of the SPC4. Switching the multiplexer is done by switching the multiplexer on and off at a rate predicted by one of the DSP serial port clocks. If we wish to switch the multiplexer at 8 kHz (1 kHz sample rate for each channel), then the DSP should be programmed to produce a 16 kHz clock signal. The speed duplication is used to switch the multiplexer off first during one clock period and on again at the next channel for the next clock period. In this way we are sure that there are never two bridges connected to the A/D-converter at the same time.

When the multiplexer is switched to the next channel, the A/D-conversion is started. The A/D-converter will send the result automatically to a serial port of the DSP after the conversion has taken place. The state machine for controlling the A/D-conversions can be found in Appendix E.
Memory management is done by looking at the value of two memory-mapped registers: *Memmode* and *Flashbank*. The possible values of *Memmode* are:

00	-	Boot mode
01	-	Normal mode
10	-	DSP-Boot/Debug mode
11	-	undefined

The upper six bits of this register are unused and always zero. The *Flashbank* register selects which flash bank is mapped in the *DSP-Boot/Debug* mode. Its value can range from 0 to 7, so only the lower three bits are used.

The upper 16kB of the data-memory is reserved for the Glue logic and its organisation is as in table 9.1.

Table 9.1: Glue Logic Memory Organisation

Address	*Function*
E000-E5FF	SPC4 dual ported RAM
F000	Memmode register
F001	Flashbank register (write-only)
F010-F017	HIP registers

According to the address and the memory mode, either the SPC4, flash, SRAM or HIP is selected. The complete VHDL-code for the Glue logic can be found in Appendix E.

9.4 80C32 Software

As mentioned previously, the 80C32 basically passes incoming commands from the Profibus to the DSP. We could therefore simply take the command set defined for the Host Interface Port and use it again for the Profibus. This would mean that the already heavily occupied master station has to take care of switching to the proper DSP runlevel and all the low-level commands. The performance of the master would degrade while the weight controller has CPU power left, both on the DSP and the 80C32. Shifting the low-level tasks to the 80C32 seems more logical. The master station can send higher-level commands like “cup number 273 is on the bridge, measure its weight”, while the 80C32 has to instruct the DSP to go to the proper runlevel and start the measurement.

In this manner, the master station has only five commands to choose from:

- *Get a curve*
- *Initialise delays, channels and sample frequency*
- *Initialise the FIR filter*
- *Start a measurement*
- *Abort all measurements*

Besides that, it can receive one of three messages from the weight controller:
- *An error message*
- *A result from a weight measurement*
- *A result from a curve measurement*

The precise command description can be found in Appendix G. One SAP is reserved for the weight controller. The first byte of the user data unit is defined to be the command or the return message. All other bytes depend on the specific command. Whenever a message for our SAP is received, an indication is set. This indication is a pointer to a *puffer* type as described in paragraph 8.2. In the main loop of the 80C32 program we call a function that checks whether or not the indication is received. If this is true, appropriate commands are send to the DSP, depending on the first byte of the data unit.

The DSP will give an interrupt each time it sends a message. This can be an acknowledge, an error or a message that weight or curve results are coming up. To keep track of the data stream coming from or going to the DSP, the 80C32 uses a state machine. In each state it knows what data it is expecting and what it has to do with it. After all data is processed, it will enter the IDLE state until a new command is received. Transmission of a frame to the master station is handled by the SPC4. The 80C32 can only use the FDL functions described in paragraph 8.2 to update the reply message in the SPC4's Dual Ported Memory. To avoid overwriting a message that has not been sent yet, the 80C32 may only update the reply immediately after receipt of an indication. At that time, the previous reply has already been send by the SPC4 and updating it is consequently safe.

10. Conclusions & Recommendations

10.1 Conclusions

We designed a weight controller that consists of the following components:
- ✓ Bridge excitation circuitry
- ✓ Bridge amplifier
- ✓ Multiplexed A/D-converter
- ✓ Digital Signal Processor
- ✓ 80C32 host-processor
- ✓ Profibus interface
- ✓ Glue logic

All of these components are described and a hardware design of the weight controller has been made. The software for both the DSP and the 80C32 has been written and simulated. A VHDL program has been written and simulated for the Glue logic in order to let all weight controller components work together.

Two methods for weight measurements were opposed. The first one is filtering the bridge signals with a FIR-filter and averaging ten filtered samples. This method was tested on an Agra machine and resulted in a weight precision of 2 grams. The second one is analysing the damping ratio and oscillation frequency of a bridge signal. This can only be done on machines with more sophisticated mechanical behaviour than the Agra machines.

The communication protocol between the DSP and the 80C32 has been implemented and simulated and the various messages have been described. The Profibus protocol has been described in detail as well, but since it is performed by the SPC4 in hardware and the weight controller's hardware is not assembled yet, it still has to be tested.

10.2 Recommendations

Future versions of the weight controller will have to be able to communicate via the Profibus-DP protocol to ensure proper installation in other Profibus-DP networks. For the same reason it is useful to implement the a-cyclic Profibus-DPE protocol, so that slaves can be polled more than once in a poll-cycle. This is necessary for the desired timing specifications as described previously.

Both weight methods have to be tested on data from a machine other than one from Agra. The desired precision of one gram might then be obtained with the "averaging" method. The method using damping ratio and oscillation frequency might give even better results.

11. REFERENCES

[1] David, C.F.L.
DESIGN OF AN ACTUATOR-CONTROLLER USING PROFIBUS-DP.
Information and Communication Systems Section, Faculty of Electrical Engineering, Eindhoven University of Technology, 1997.

[2] Popp, M.
THE RAPID WAY TO PROFIBUS-DP.
Profibus Nutzerorganisation, 1997.

[3] TMS320C54X DSP APPLICATIONS GUIDE, Preliminary Revision.
Texas Instruments Incorporated, 1996.

[4] DIGITAL SIGNAL PROCESSING APPLICATIONS USING THE ADSP-2100 FAMILY, Volume 1.
New Jersey: Prentice Hall, 1992.

[5] ADSP-2100 FAMILY USER'S MANUAL, 3rd edition.
Norwood: Analog Devices Incorporated, September 1995.

[6] ADS7809, 16-BIT 10μs SERIAL CMOS SAMPLING ANALOG-TO-DIGITAL CONVERTER.
Tucson: Burr-Brown Corporation, 1996.

[7] Franklin, G.F. and J.D. Powell, A. Emami-Naeini
FEEDBACK CONTROL OF DYNAMIC SYSTEMS, 3rd edition.
New York: Addison-Wesley, 1994.

[8] Borghouts, A.N.
INLEIDING IN DE MECHANICA, 3rd edition.
Delft: Delftse Uitgevers Maatschappij, 1976.

[9] Ritzerfeld, J. and H. Hegt, P. Sommen, H. van Meer
REALISERING VAN DIGITALE SIGNAALBEWERKENDE SYSTEMEN.
Faculty of Electrical Engineering, Eindhoven University of Technology, 1993.
Collegedictaat nr. 5750.

[10] DSP MICROCOMPUTER ADSP-2171/ADSP-2172/ADSP-2173, Revision A.
Norwood: Analog Devices Incorporated, 1995.

12. ACKNOWLEDGEMENT

I would like to thank everyone at Ellips for their help and support during my graduation period. In particular Ivo Kleuters for his help with test equipment and demolishing the Agra-machine, Harry Stox for his Profibus and software hints and tips, Jean-Paul Smeets for all the hardware clues and suggestions, Sandra de Groot for several administrational jobs and the cookies, Erwin Bakker for the graduation place, Camile David for his work on the I/O-card, Tiago Gons for the racing game (without fun no quality!) and Roland Scheffer for his miraculous soldering work and Orcad-knowledge.

Furthermore I would like to thank Ronnie te Lintelo from Eurodis/Texim Electronics for his help with choosing hardware components and delivering samples without delay and professor Stevens for his comments on this graduation work.

APPENDIX A: NOMENCLATURE

A/D-converter - Analog to Digital converter, converts analogue signals to discrete digital levels.
ALU - Arithmetic and Logical Unit, provides a standard set of arithmetic and logical functions such as add, subtract, negate, AND, OR, NOT, etc.
ASIC - Application Specific Integrated Circuit.
Digital Signal Processor - Microprocessor optimised for digital signal processing and other high-speed numeric processing applications.
DSP - See *Digital Signal Processor.*
FC - Frame Control octet, indicates the frame type, the function and control information to prevent loss and multiplication of messages.
FDL - Fieldbus Data Link layer, layer 2 of the Profibus model.
FPGA - Field Programmable Gate Array, array of logic cells that communicate with one another and with I/O via wires within routing channels.
Frame - Sequence of UARTs that conveys a message.
GAP - The address range between This Station (TS) and the address of the Next Station (NS). See also *TS* and *NS.*
GAPL - GAP List, list of addresses of all slave stations in the GAP.
HDR - HIP data register, one of the memory locations of the Host Interface Port. See also *HIP* and *HSR.*
HIP - Host Interface Port, parallel I/O port that allows a processor to be used as memory-mapped peripheral of a host computer. See also *HDR* and *HSR.*
HSR - Host Status Register, one of the status registers of the Host Interface Port. See also *HIP* and *HDR.*
MAC - Multiplier/Accumulator, provides high-speed multiplication, multiplication with cumulative addition, multiplication with cumulative subtraction, saturation and clear-to-zero functions.
NS - Next Station, the master station to which the token will be transmitted.
Poll List - List that indicates in which order the slaves will be polled by the master station.
Profibus - Process Field Bus, German field bus standard
Profibus-DP - Decentralised Periphery, Profibus standard for high-speed data communication required in factory automation and building automation.
PS - Previous Station, the master station from which the token is received.
SD1-frame - Frame of fixed length without a data field.
SD2-frame - Frame with variable data field length.
SD3-frame - Frame with fixed data field length.
Sport - Serial port of the Digital Signal Processor, Sport0 refers to serial port zero and Sport1 refers to serial port 1.
$T_{A/R}$ - Transmission time of the response frame.
T_{ENC} - Time between two polls of the encoder.
T_{ID} - Idle Time, time that elapses between response and new request.
T_{MC} - Message Cycle Time, the time that elapses between two consecutive transmissions of a Profibus master station.
T_{SDR} - Station Delay Time of Responders, time that elapses between request and

response.

$T_{S/R}$ - Transmission time of the action frame.

T_{SR} - System Reaction Time, the amount of time needed to poll all slaves.

T_{TD} - Transmission Delay Time, time that elapses on the transmission medium between transmitter and receiver when a frame is transmitted.

Token - Symbolic indication to denote the master that has the right of access to the Profibus.

TS - This Station, the address of the master station under consideration.

UART-character - Universal Asynchronous Receiver/Transmitter character.

VHDL - Very high speed integrated circuit Hardware Description Language, language to describe and build hardware logic.

APPENDIX B: PROFIBUS PERFORMANCE

Equations:

$$T_{SR} = T_{MC,TOTAL} + mp \bullet T_{MC,RET}$$

$$T_{MC} = T_{S/R} + T_{SDR} + T_{A/R} + T_{ID} + 2 \cdot T_{TD}$$

$$T_{MC,\ TOTAL} = nr \cdot T_{MC,RELAY} + T_{MC,WEIGHT} + \left\lceil \frac{np}{n} \right\rceil \cdot T_{MC,ENCODER}$$

$$T_{ENC} = \frac{T_{SR}}{\left\lceil \frac{np-1}{n} \right\rceil}$$

Assume:

$$T_{MC,RET} = T_{MC,RELAY}$$

$$mp = \begin{cases} 1 & \text{for 187.5 kbit / s, 500 kbit / s and 1.5 Mbit / s} \\ 2 & \text{for 3 Mbit / s} \\ 3 & \text{for 6 Mbit / s} \\ 4 & \text{for 12 Mbit / s} \end{cases}$$

$$nr = 28$$
$$np = 30$$

$$T_{S/R} = \begin{cases} \text{143 bits} & \text{for the relay controller and the weight controller} \\ \text{66 bits} & \text{for the encoder} \end{cases}$$

$$T_{A/R} = \begin{cases} \text{11 bits} & \text{for the relay controller} \\ \text{143 bits} & \text{for the encoder} \\ \text{319 bits} & \text{for the weight controller} \end{cases}$$

Description:

This simulated Profibus system consists of one master and 30 slaves of which one is a weight controller, one an encoder and 28 are I/O-cards. The Poll List of the master station is constructed in such a way that it will poll *n* slaves, then the encoder, *n* other slaves, the encoder again, and so on. Decreasing *n* will enhance the encoder resolution, but reduce the System Reaction Time T_{SR}. Tables B1 and B2 on the next page show calculations for several values of *n* and different bit rates. Greyed values are acceptable for the fruit grading system.

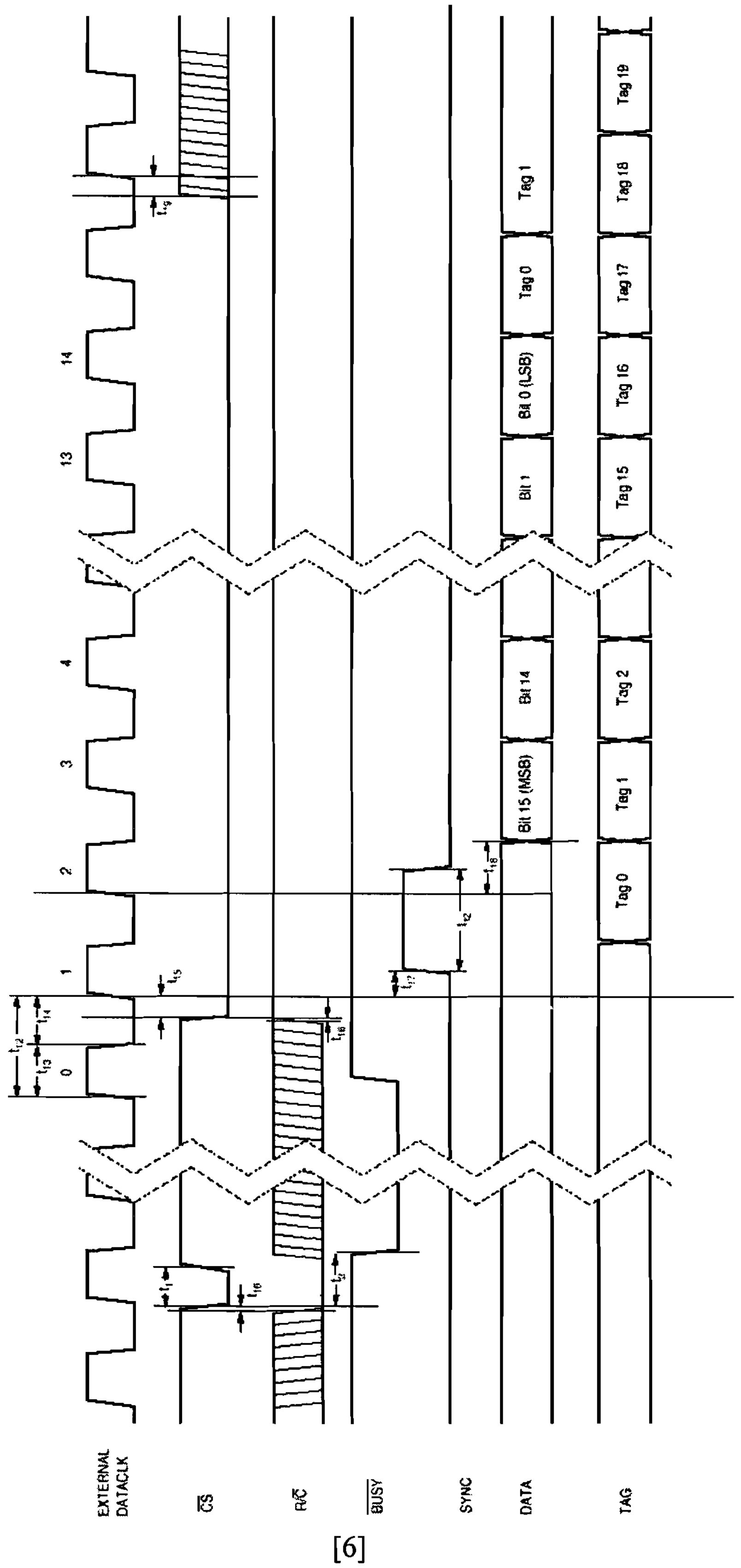
Table B1: System Reaction Time T_{SR} (ms)

	187.5 kbit/s	*500 kbit/s*	*1.5 Mbit/s*	*3 Mbit/s*	*6 Mbit/s*	*12 Mbit/s*
n=1	100.4	47.2	17.8	10.0	7.2	5.5
n=3	64.5	30.6	11.6	6.6	4.8	3.7
n=5	57.4	27.3	10.3	5.9	4.3	3.4
n=8	53.8	25.6	9.7	5.6	4.1	3.2
n=10	52.0	24.8	9.4	5.4	4.0	3.1
n=15	50.2	23.9	9.1	5.2	3.8	3.0

Table B2: Encoder Resolution T_{ENC} (ms)

	187.5 kbit/s	*500 kbit/s*	*1.5 Mbit/s*	*3 Mbit/s*	*6 Mbit/s*	*12 Mbit/s*
n=1	3.5	1.6	0.6	0.3	0.2	0.2
n=3	6.5	3.1	1.2	0.7	0.5	0.4
n=5	9.6	4.5	1.7	1.0	0.7	0.6
n=8	13.4	6.4	2.4	1.4	1.0	0.8
n=10	17.3	8.3	3.1	1.8	1.3	1.0
n=15	25.1	12.0	4.5	2.6	1.9	1.5

APPENDIX C: ADS7809 TIMING CONDITIONS



[6]

ADS7809 Timing Specifications

SYMBOL	DESCRIPTION	MIN	TYP	MAX	UNITS
t_1	Convert Pulse Width	40		6000	ns
t_2	$\overline{\text{BUSY}}$ Delay			65	ns
t_3	$\overline{\text{BUSY}}$ LOW			8	µs
t_4	$\overline{\text{BUSY}}$ Delay after End of Conversion		220		ns
t_5	Aperture Delay		40		ns
t_6	Conversion Time		7.6	8	µs
t_7	Acquisition Time			2	µs
$t_6 + t_7$	Throughput Time		9	10	µs
t_8	R/$\overline{\text{C}}$ LOW to DATACLK Delay		450		ns
t_9	DATACLK Period		440		ns
t_{10}	Data Valid to DATACLK HIGH Delay	20	75		ns
t_{11}	Data Valid after DATACLK LOW Delay	100	125		ns
t_{12}	External DATACLK	100			ns
t_{13}	External DATACLK HIGH	20			ns
t_{14}	External DATACLK LOW	30			ns
t_{15}	DATACLK HIGH Setup Time	20		t_{12} +5	ns
t_{16}	R/$\overline{\text{C}}$ to $\overline{\text{CS}}$ Setup Time	10			ns
t_{17}	SYNC Delay After DATACLK HIGH	15		35	ns
t_{18}	Data Valid Delay	25		55	ns
t_{19}	$\overline{\text{CS}}$ to Rising Edge Delay	25			ns
t_{20}	Data Available after $\overline{\text{CS}}$ LOW	6			µs

APPENDIX D: FIR FILTER SOURCE

```
/*
        This function computes the (FIR) filtered response of a given
        input sample. The function returns the filtered value of the input.

        H. Kester, June 11, 1997

        - uses non-circular state-buffer

        int firfilt(int sample, int coeffs[], int state[], int taps);
*/

#include <asm_sprt.h>


.MODULE         __firfilt__;

.ENTRY          firfilt_;

firfilt_:       function_entry;         /* Function prologue           */
                save_reg;               /* Save registers              */

                MR0=AR;                 /* Hold input sample           */
                MR1=AY1;                /* Hold coeff pointer          */

                readsfirst(SR0);        /* Fetch pointer to state array */
                I0=SR0;                 /* Store for calculations      */
                AY1=readsnext;          /* Fetch number of TAPS        */

                AR=AY1;                 /* Load number of TAPS         */
                AR=SR0+AY1;             /* Calculate pointer address   */
                I1=AR;                  /* Store for reading           */

                I6=I1;                  /* Hold address of state ptr   */
                AR=DM(I1, M1);          /* Load state pointer          */
                SR1=MX0, AR=PASS AR;    /* Test for first invocation   */
                IF EQ AR=PASS SR0;      /* Point to beginning of array */

                I1=AR;                  /* Point to current location   */
                DM(I1, M1)=MR0;         /* Place input into delay line */

                AY0=I6;                 /* Load address of state ptr   */
                AR=I1;                  /* Load current pointer        */
                AR=AR-AY0;              /* Are they equal?             */
                AR=I1;                  /* If not, still use old ptr   */
                IF EQ AR=PASS SR0;      /* Else, point to beginning    */
                MR0=AR;                 /* Copy pointer for write      */
                AR=AY1-1, DM(I6,M5)=MR0;/* Write new pointer into state */
                I6=MR1;                 /* Point to coeff array        */
                SR0=MSTAT;              /* Save mode for later         */
                DIS M_MODE;             /* Enter integer mode          */
                M5=1;                   /* Set to proper value         */
                MR=0, MX0=DM(I0,M1), MY1=PM(I6,M5);/* Load delay, coeff */
                CNTR=AR;                /* Loop TAPS-1                 */
                DO __convolution UNTIL CE;
__convolution:  MR=MR+MX0*MY1 (SS), MX0=DM(I0,M1), MY1=PM(I6,M5);
                MR=MR+MX0*MY1 (RND);
                IF MV SAT MR;
                MX0=SR1, AR=PASS MR1;   /* Restore MX0 for return      */
                MSTAT=SR0;              /* Restore old mode            */
                restore_reg;            /* Restore registers           */
                exit;                   /* Function epilogue           */

.ENDMOD;
```

APPENDIX E: VHDL SOURCE

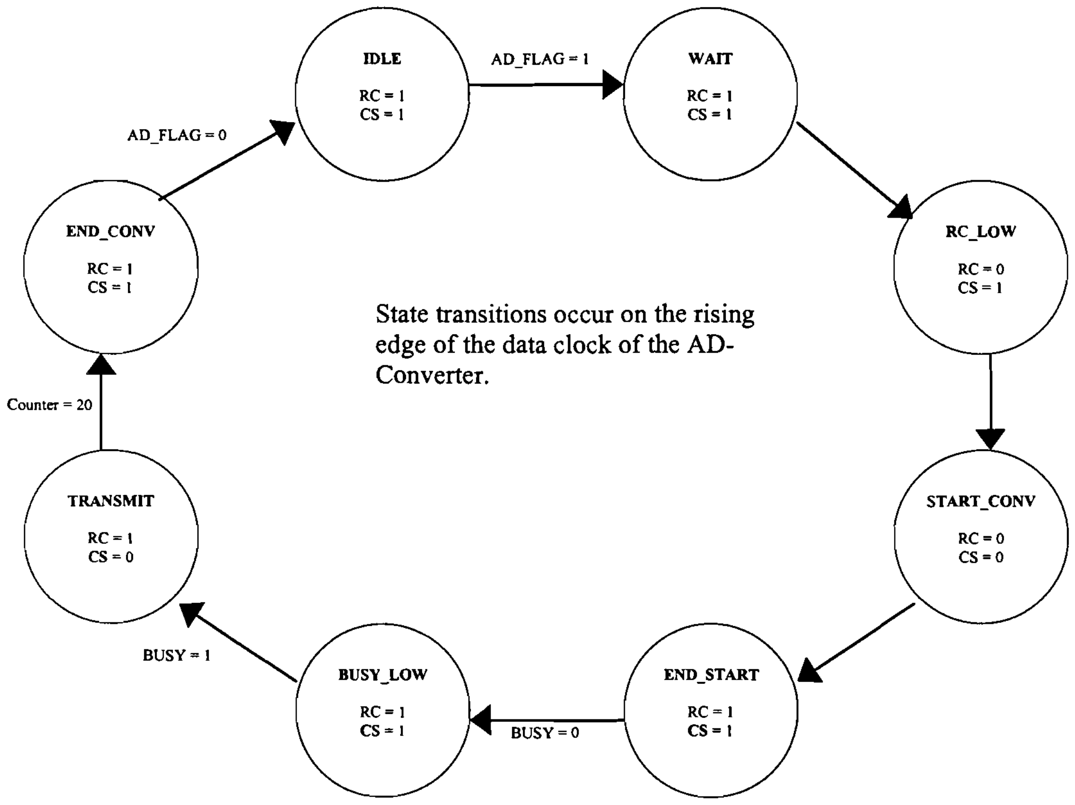


Figure E1: State Machine for AD-Converter Control

```
--- Profibus weight card glue logic
---
--- Revision history
--- 14-08-1997 first version
--- 21-08-1997 addition of MMU


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-------------------------------------------------------------------------------------

entity gwmemdec is
        port(
                ale             :       in std_logic;
                psen            :       in std_logic;
                rd_inn          :       in std_logic;
                wrn             :       in std_logic;
                clk_in          :       in std_logic;
                sample_clk      :       in std_logic;
                ad_dclk         :       in std_logic;
                rst_80c32       :       in std_logic;
                sync            :       in std_logic;
                ad_busyn        :       in std_logic;
                addr_in_7_0     :       inout std_logic_vector(7 downto 0);
```

```
              addr_in_15_12   :       in std_logic_vector(15 downto 12);
              addr_out_7_0    :       out std_logic_vector(7 downto 0);
              addr_out_16_14  :       out std_logic_vector(16 downto 14);
              rd_outn         :       out std_logic;
              flash_csn       :       out std_logic;
              ram_csn         :       out std_logic;
              spc4_csn        :       out std_logic;
              hseln           :       out std_logic;
              ad_csn          :       out std_logic;
              ad_rc           :       out std_logic;
              mux_out         :       out std_logic_vector(7 downto 0);
              clk_dsp         :       out std_logic
        );
end gwmemdec;

-------------------------------------------------------------------------------------

architecture memdec_arch of gwmemdec is

  type mux_state_type is (channel0_off, channel0_on,channel1_off, channel1_on,
                          channel2_off, channel2_on,channel3_off, channel3_on,
                          channel4_off, channel4_on,channel5_off, channel5_on,
                          channel6_off, channel6_on,channel7_off, channel7_on);
  type ad_state_type is (ad_idle, ad_wait, ad_rc_low, ad_start_conv, ad_end_start,
                          ad_busy_low, ad_transmit, ad_end_conv);

  constant spc4_base: std_logic_vector (15 downto 0) := X"E000";
  constant glue_base: std_logic_vector (15 downto 0) := X"F000";


  -- local signals
  signal latched_addr_7_0 : std_logic_vector(7 downto 0);
  signal memmode : std_logic_vector(1 downto 0);
  signal flash_bank : std_logic_vector(2 downto 0);
  signal mux_present_state, mux_next_state : mux_state_type;
  signal ad_present_state, ad_next_state : ad_state_type;
  signal count_clk_dsp : std_logic;
  signal ad_flag : std_logic;
  signal count_ad : std_logic_vector (4 downto 0);

begin

  -- mux outputs
  mux_state_proc: process (mux_present_state)
  begin
   case mux_present_state is
     when channel0_off =>
        ad_flag <= '0';
        mux_out <= "00000000";
        mux_next_state <= channel0_on;
     when channel0_on =>
        ad_flag <= '1';
        mux_out <= "00000001";
        mux_next_state <= channel1_off;
     when channel1_off =>
        ad_flag <= '0';
        mux_out <= "00000000";
        mux_next_state <= channel1_on;
     when channel1_on =>
        ad_flag <= '1';
        mux_out <= "00000010";
        mux_next_state <= channel2_off;
     when channel2_off =>
        ad_flag <= '0';
        mux_out <= "00000000";
        mux_next_state <= channel2_on;
     when channel2_on =>
        ad_flag <= '1';
        mux_out <= "00000100";
        mux_next_state <= channel3_off;
     when channel3_off =>
        ad_flag <= '0';
```

```
          mux_out <= "00000000";
          mux_next_state <= channel3_on;
       when channel3_on =>
          ad_flag <= '1';
          mux_out <= "00001000";
          mux_next_state <= channel4_off;
       when channel4_off =>
          ad_flag <= '0';
          mux_out <= "00000000";
          mux_next_state <= channel4_on;
       when channel4_on =>
          ad_flag <= '1';
          mux_out <= "00010000";
          mux_next_state <= channel5_off;
       when channel5_off =>
          ad_flag <= '0';
          mux_out <= "00000000";
          mux_next_state <= channel5_on;
       when channel5_on =>
          ad_flag <= '1';
          mux_out <= "00100000";
          mux_next_state <= channel6_off;
       when channel6_off =>
          ad_flag <= '0';
          mux_out <= "00000000";
          mux_next_state <= channel6_on;
       when channel6_on =>
          ad_flag <= '1';
          mux_out <= "01000000";
          mux_next_state <= channel7_off;
       when channel7_off =>
          ad_flag <= '0';
          mux_out <= "00000000";
          mux_next_state <= channel7_on;
       when channel7_on =>
          ad_flag <= '1';
          mux_out <= "10000000";
          mux_next_state <= channel0_off;
 end case;
end process;

-- state transition for mux
mux_state_change: process (sample_clk, sync)
begin
 if sync = '0' then -- mux reset
     mux_present_state <= channel0_off;
 else
     if (sample_clk'event and sample_clk = '1') then
        mux_present_state <= mux_next_state;
     end if;
 end if;
end process;

-- control A/D-conversions
ad_state_proc: process(ad_present_state, ad_busyn, ad_flag, count_ad)
begin
 case ad_present_state is
   when ad_idle =>
    ad_rc <= '1';
    ad_csn <= '1';
    if ad_flag = '1' then
       ad_next_state <= ad_wait;
    else
        ad_next_state <= ad_idle;
    end if;
   when ad_wait =>
    ad_rc <= '1';
    ad_csn <= '1';
    ad_next_state <= ad_rc_low;
   when ad_rc_low =>
    ad_rc <= '0';
    ad_csn <= '1';
```

```
     ad_next_state <= ad_start_conv;
    when ad_start_conv =>
     ad_rc <= '0';
     ad_csn <= '0';
     ad_next_state <= ad_end_start;
    when ad_end_start =>
     ad_rc <= '1';
     ad_csn <= '1';
     if ad_busyn = '0' then
        ad_next_state <= ad_busy_low;
     else
        ad_next_state <= ad_end_start;
     end if;
    when ad_busy_low =>
     ad_rc <= '1';
     ad_csn <= '1';
     if ad_busyn = '1' then
        ad_next_state <= ad_transmit;
     else
        ad_next_state <= ad_busy_low;
     end if;
    when ad_transmit =>
     ad_rc <= '1';
     ad_csn <= '0';
     if count_ad = "10100" then -- wait 20 ad_dclk cycles
        ad_next_state <= ad_end_conv;
     else
        ad_next_state <= ad_transmit;
     end if;
    when ad_end_conv =>
     ad_rc <= '1';
     ad_csn <= '1';
     if ad_flag = '0' then
        ad_next_state <= ad_idle;
     else
        ad_next_state <= ad_end_conv;
     end if;
 end case;
end process;

-- ad counter
ad_count_proc: process (ad_dclk, ad_present_state, sync)
begin
 if sync = '0' then
    count_ad <= "00000";
 else
    if (ad_dclk'event and ad_dclk = '1') then
       case ad_present_state is
         when ad_transmit => count_ad <= count_ad + 1;
         when others      => count_ad <= "00000";
       end case;
    end if;
 end if;
end process;

-- ad state transitions
ad_state_change: process (ad_dclk, sync)
begin
 if sync = '0' then
    ad_present_state <= ad_idle;
 else
    if (ad_dclk'event and ad_dclk = '1') then
      ad_present_state <= ad_next_state;
    end if;
 end if;
end process;

-- DSP clock must run at half the clk_in rate
clock_dsp_proc: process (clk_in, rst_80c32)
begin
  if rst_80c32 = '1' then
    count_clk_dsp <= '0';
  else
    if (clk_in'event and clk_in = '1') then
      count_clk_dsp <= not count_clk_dsp;
    end if;
  end if;
end process;

clk_dsp <= '1' when count_clk_dsp = '0' else '0';


-- Memory control -----------------------------------------------------

-- address latch
addr_latch_proc: process (ale, addr_in_7_0)
begin
 if ale = '1' then
    latched_addr_7_0 <= addr_in_7_0;
 end if;
end process;

addr_out_7_0 <= latched_addr_7_0;

-- assert rd_out if either psen or rd_in is asserted
rd_outn <= not (not psen or not rd_inn);

-- memmode write
memmode_wr_proc: process (wrn, rst_80c32)
begin
 if rst_80c32 = '1' then
    memmode <= "00";
 else
    if (wrn'event and wrn = '0') then
       if ((addr_in_15_12 = glue_base(15 downto 12)) and
                            (latched_addr_7_0(4 downto 0) = "00000")) then
          memmode <= addr_in_7_0(1 downto 0);
       end if;
    end if;
 end if;
end process;

-- memmode read
memmode_rd_proc: process (rd_inn)
begin
 if (rd_inn'event and rd_inn = '0') then
    if ((addr_in_15_12 = glue_base(15 downto 12)) and
            (latched_addr_7_0(4 downto 0) = "00000")) then
       addr_in_7_0 <= "000000" & memmode;
    else
       addr_in_7_0 <= "ZZZZZZZZ";
    end if;
 end if;
end process;

-- flash bank write
flashbank_proc: process (wrn, rst_80c32)
begin
 if rst_80c32 = '1' then
    flash_bank <= "000";
 else
    if (wrn'event and wrn = '1') then
       if ((addr_in_15_12 = glue_base(15 downto 12)) and
             (latched_addr_7_0(4 downto 0) = "00001")) then
          flash_bank <= addr_in_7_0(2 downto 0);
       end if;
    end if;
 end if;
end process;

-- chip select control
select_proc: process (psen, addr_in_15_12, memmode, flash_bank, latched_addr_7_0)
begin
 if psen = '0' then -- program memory
```

```
      case memmode is
        when "00" => -- lower 64K of flash
         addr_out_16_14 <= '0'&addr_in_15_12(15 downto 14);
         flash_csn <= '0';
         ram_csn <= '1';
         spc4_csn <= '1';
         hseln <= '1';
        when "01" => -- upper 64K of flash
         addr_out_16_14 <= '1'&addr_in_15_12(15 downto 14);
         flash_csn <= '0';
         ram_csn <= '1';
         spc4_csn <= '1';
         hseln <= '1';
        when "10" => -- 32K RAM only
         addr_out_16_14 <= '-'&addr_in_15_12(15 downto 14);
         flash_csn <= '1';
         ram_csn <= '0';
         spc4_csn <= '1';
         hseln <= '1';
        when "11" => -- illegal mode
         addr_out_16_14 <= '0'&addr_in_15_12(15 downto 14);
         flash_csn <= '1';
         ram_csn <= '1';
         spc4_csn <= '1';
         hseln <= '1';
      end case;
   else -- data memory
      if addr_in_15_12(15) = '0' then -- lower 32K is RAM
         addr_out_16_14 <= '-'&addr_in_15_12(15 downto 14);
         flash_csn <= '1';
         ram_csn <= '0';
         spc4_csn <= '1';
         hseln <= '1';
      else
         if addr_in_15_12(14) = '1' then -- upper 16K is Glue
            addr_out_16_14 <= "---";
            flash_csn <= '1';
            ram_csn <= '1';
            if addr_in_15_12 = spc4_base(15 downto 12) then
               spc4_csn <= '0';
            else
               spc4_csn <= '1';
            end if;
            if ((addr_in_15_12 = glue_base(15 downto 12)) and
                     (latched_addr_7_0(4)='1')) then
               hseln <= '0';
            else
               hseln <= '1';
            end if;
         else  -- flash area
            if memmode = "10" then -- flash only mapped when in mode "01"
               addr_out_16_14 <= flash_bank;
               flash_csn <= '0';
               ram_csn <= '1';
               spc4_csn <= '1';
               hseln <= '1';
            else -- illegal mode
               addr_out_16_14 <= "---";
               flash_csn <= '1';
               ram_csn <= '1';
               spc4_csn <= '1';
               hseln <= '1';
            end if;
         end if;
      end if;
   end if;
  end process;
```

end memdec_arch;

APPENDIX F: HIP COMMAND OVERVIEW

```
/* Messages from and to the DSP ***********************************************/

/* DSP runlevels ***************************************************************/
#define RUN_BOOTED              0       /* DSP has completed booting          */
#define RUN_GONE                1       /* DSP is running loaded program      */
#define RUN_INIT                2       /* DSP is initializing                */
#define RUN_FILTERING           3       /* DSP is filtering data              */
#define RUN_MEASURING           4       /* DSP is measuring weights           */


/* Host (80c32) commands ******************************************************/

/* Operation:
    1. Set command in command register of hip-structure
    2. Wait for HIP_ACK
    3. Set data in data registers of hip-structure
    4. Set NEW_DATA in command register of hip structure
    Repeat step 2,3 and 4 until all data is transmitted.

    DSP interrupt is set on writing the command register.
*/

#define COM_GO                  0x00    /* Start program                      */

#define COM_INIT                0x10    /* Switch to INIT runlevel            */
#define COM_GET_RAW_CURVE       0x11    /* Get curve of raw data              */
/* parameters
    host_data1 channel_nr               [1...8]

    host_data1 MSB sample frequency
    host_data2 LSB sample frequency

    host_data1 MSB number of points
    host_data2 LSB number of points
*/
#define COM_STOP_GONE           0x12    /* Switch to BOOTED runlevel          */

#define COM_SET_DELAY           0x20    /* Set measurement delay              */
/* parameters
    host_data1 MSB delay_value channel 1
    host_data2 LSB delay_value channel 1    (1 LSB is 100 microseconds)

    host_data1 MSB delay_value channel 2
    host_data2 LSB delay_value channel 2

    ...
    ...

    host_data1 MSB delay_value channel 8
    host_data2 LSB delay_value channel 8
*/
#define COM_SET_CHANNELS        0x21    /* Select weight channels             */
/* parameters
    host_data1 channel_mask     bit 0 = channel 1, OFF=0 ON=1
                                bit 7 = channel 8
*/
#define COM_SET_SAMPLE_FREQ     0x22    /* Set sample frequency for one channel */
/* parameters
    host_data1 MSB of frequency
    host_data2 LSB of frequency
*/
#define COM_SET_FILTER_LEN      0x23    /* Set filter length                  */
/* parameters
    host_data1 filter_len
*/
#define COM_SET_SAMPLES         0x24    /* Set number of samples for averaging  */
/* parameters
    host_data1 number_of_samples
*/
#define COM_STOP_INIT           0x25    /* Stop initializing filter           */
#define COM_START_FILTERING     0x26    /* Start filtering on all channels    */
#define COM_SET_FILTER_CONST    0x27    /* Set filter constants               */
/* parameters
    host_data1 MSB filter_constant1
    host_data2 LSB filter_constant1

    host_data1 MSB filter_constant2
    host_data2 LSB filter_constant2

    ...
    ...

    host_data1 MSB last_filter_constant
    host_data2 LSB last_filter_constant
*/

#define COM_START_MEASUREMENTS  0x30    /* Start weight measurements on selected channels */
/* parameters
    host_data1 MSB cup_number
    host_data2 LSB cup_number
*/
```

```
#define COM_STOP_FILTERING      0x31    /* Stop filtering data             */
#define COM_GET_FILTERED_CURVE  0x32    /* Get curve of filtered data      */
/* parameters
    host_data1 channel_nr

    host_data1 MSB sample frequency
    host_data2 LSB sample frequency

    host_data1 MSB number of points
    host_data2 LSB number of points
*/

#define COM_STOP_MEASUREMENTS   0x40    /* Stop all weight measurements    */

#define COM_GET_RUNLEVEL        0x7F    /* Get current DSP runlevel        */
#define NEW_DATA                0xFE    /* New data has arrived            */
#define HIP_ACK                 0xFF    /* Acknowledge that previous command or */
                                        /*  data has been received         */


/* DSP messages ****************************************************************/

/*
   Operation:
   1. Confirm receipt of message with HIP_ACK in command register of hip-structure
   2. Wait for NEW_DATA in dsp_status register
   3. Read data in dsp_data1 and dsp_data2 registers
   4. Send HIP_ACK to command register
   Repeat step 2,3 and 4 until all data is transmitted

*/

#define DSP_OK                  0xD0    /* No messages                      */
#define DSP_WEIGHTS             0xD1    /* Measured weights of selected channels */
/* data
    dsp_data1 MSB cup_number
    dsp_data2 LSB cup_number

    dsp_data1 MSB weight lowest selected channel
    dsp_data2 LSB weight lowest selected channel

    dsp_data1 MSB weight second lowest selected channel
    dsp_data2 LSB weight second lowest selected channel

    ...
    ...

    dsp_data1 MSB weight highest selected channel
    dsp_data2 LSB weight highest selected channel
*/

#define DSP_CURVE               0xD2    /* Curve of requested channel      */
                                        /* Filtered or not filtered        */
/* data
    dsp_data1 MSB channel_nr
    dsp_data2 LSB channel_nr

    dsp_data1 MSB sample frequency
    dsp_data2 LSB sample frequency

    dsp_data1 MSB number of points
    dsp_data2 LSB number of points

    dsp_data1 MSB dataword1
    dsp_data2 LSB dataword1

    dsp_data1 MSB dataword2
    dsp_data2 LSB dataword2

    ...
    ...

    dsp_data1 MSB last_dataword
    dsp_data2 LSB last_dataword
*/


#define DSP_RUNLEVEL            0xD3    /* Current DSP runlevel            */
/* data
    dsp_data1 MSB dsp_runlevel
    dsp_data2 LSB dsp_runlevel
*/


/* DSP error messages **********************************************************/

#define DSP_NOT_ALLOWED         0xE0    /* Command not allowed for this runlevel */
#define DSP_NOT_FINISHED        0xE1    /* Previous command not finished yet     */
#define DSP_NO_COMMAND          0xE2    /* No command specified yet              */
#define DSP_FILT_TOO_LONG       0xE3    /* Requested filter length is too long   */
#define DSP_INVALID_FREQ        0xE4    /* Invalid sample frequency requested    */
#define DSP_INVALID_CURVE_LEN   0xE5    /* Invalid curve length requested        */
#define DSP_INVALID_CHANNEL     0xE6    /* Invalid channel requested             */
#define DSP_NO_CHANNEL          0xE7    /* No channel selected                   */
#define DSP_TOO_MANY_JOBS       0xE8    /* Too many measurement jobs in progress */
```

```
/* Structures ****************************************************************/

typedef struct hipst
{
 uchar command;
 uchar host_data1;
 uchar host_data2;
 uchar dsp_status;
 uchar dsp_data1;
 uchar dsp_data2;
 uchar hip_status_reg6;
 uchar hip_status_reg7;
} hipt;
```

APPENDIX G: PROFIBUS COMMAND OVERVIEW

```
/* Profibus messages from and to the weight controller ********************/

#define RAW       0          /* Raw curve      */
#define FILTERED  1          /* Filtered curve */

/************************************************************************
   Commands for the weight controller

   Operation:
    The first byte of the data unit (byte 0) of a frame is the command.
    All other bytes depend on the specific command.
************************************************************************/

#define PCOM_GET_CURVE          0         /* Request a curve            */
/* parameters
    byte 1 = packet number (1..N)
    byte 2 = channel number (1..8)
    byte 3 = type (RAW, FILTERED)
    byte 4 = MSB sample frequency
    byte 5 = LSB sample frequency
    byte 6 = MSB number of points
    byte 7 = LSB number of points
*/

#define PCOM_INIT              1          /* Setup measurement parameters  */
/* parameters
    byte 1  = MSB delay of channel 1  (1 LSB = 100 microseconds)
    byte 2  = LSB delay of channel 1
    ...
    ...
    byte 15 = MSB delay of channel 8
    byte 16 = LSB delay of channel 8
    byte 17  = channel mask, bit0 = channel 1 (OFF=0, ON=1), bit7 = channel 8
    byte 18 = MSB of sample frequency
    byte 19 = LSB of sample frequency
*/

#define PCOM_INIT_FILTER        2         /* Setup FIR filter           */
/* parameters
    byte 1  = total number of packets (1..N)
    byte 2  = packet number of this packet (1..N)

    if (packet_number==1)
      byte 3  = filter length (1..241)
      byte 4  = number of samples for averaging
      byte 5  = MSB filter constant 1 (1.15 format !)
      byte 6  = LSB filter constant 1
      byte 7  = MSB filter constant 2
      byte 8  = LSB filter constant 2
      ...
      ...
      byte 19 = MSB filter constant 8
      byte 20 = LSB filter constant 8

    if (packet_number>1)
      byte 3  = MSB filter constant (packet_number)*9+9
      byte 4  = LSB filter constant (packet_number)*9+9
      byte 5  = MSB filter constant (packet_number)*9+10
      byte 6  = LSB filter constant (packet_number)*9+10
      ...
      ...
      byte 19 = MSB filter constant (packet_number)*9+17
      byte 20 = LSB filter constant (packet_number)*9+17
*/

#define PCOM_MEASURE            3         /* Start measurement          */
/* parameters
    byte 1  = MSB cup number
    byte 2  = LSB cup number
*/

#define PCOM_ABORT              4         /* Abort all measurements     */
/* parameters
    none
/*


/************************************************************************
   Messages from the weight controller

   Operation:
    The first byte of the data unit (byte 0) of the frame is the message.
    All other bytes depend on the specific message.
************************************************************************/

#define WEIGHT_ERROR            0x80      /* Weight controller error      */
/* data
    byte 1  = DSP error code, see "MESSAGE.H" or
                UNKNOWN_COMMAND (see below)
                ILLEGAL_LENGTH  (see below)
*/
```

```
#define WEIGHT_CURVE            0x81       /* Curve data                   */
/* data
    byte 1  = total number of packets [1..N]
    byte 2  = packet number of this packet [1..N]

    if (packet number == 1)
      byte 3  = MSB channel number
      byte 4  = LSB channel number
      byte 5  = MSB sample frequency
      byte 6  = LSB sample frequency
      byte 7  = MSB number of points
      byte 8  = LSB number of points
      byte 9  = MSB data word 1
      byte 10 = LSB data word 1
      byte 11 = MSB data word 2
      byte 12 = LSB data word 2
      ...
      ...
      byte 19 = MSB data word 6
      byte 20 = LSB data word 6

    if (packet number > 1)
      byte 3  = MSB data word (packet_number-1)*9+7
      byte 4  = LSB data word (packet_number-1)*9+7
      byte 5  = MSB data word (packet_number-1)*9+8
      byte 6  = LSB data word (packet_number-1)*9+8
      ...
      ...
      byte 19 = MSB data word (packet_number-1)*9+15
      byte 20 = LSB data word (packet_number-1)*9+15
*/

#define WEIGHT_WEIGHTS          0x82       /* Weight data                  */
/* data
    byte 1  = MSB cup number
    byte 2  = LSB cup number
    byte 3  = MSB weight lowest selected channel
    byte 4  = LSB weight lowest selected channel
    byte 5  = MSB weight second lowest selected channel
    byte 6  = LSB weight second lowest selected channel
    ...
    ...
    byte n-1= MSB weight highest selected channel
    byte n  = LSB weight highest selected channel
*/

/* Additional error messages (see WEIGHT_ERROR) ***************************/
#define UNKNOWN_COMMAND         0xFF       /* Unknown command in frame      */
#define ILLEGAL_LENGTH          0xFE       /* Wrong number of bytes in frame*/
```