EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN/PS/BD/Note 93-5

DSP SOFTWARE FOR THE PS FFT Q-MEASUREMENT SYSTEM

S. Johnston

ABSTRACT

The PS FFT Q-measurement system can perform rapid Digital Signal Processing (DSP) of pick-up data, allowing Q-calculations, FFT and Sliding FFT analysis. High processing performance is achieved in such a system by writing software which exploits the DSP hardware, and as a result, DSP software is often incompatible with other systems. This document serves as a guide to the DSP programming of the PS Q-measurement system, which is a mixture of very specific code designed for the DSP hardware and ordinary C code. Some attempt is made to explain the functionality of the specific code in order to aid transportation to other DSP systems.

Geneva, Switzerland
September 1993

CONTENTS

1. INTRODUCTION

The development of a Fast Fourier Transform (FFT) Q-measurement system to replace the swept-filter analyser [1] of the PS accelerator began in 1990 [2] with the selection of a VME-based Digital Signal Processing (DSP) card, the VASP-16 Vector & Scalar Processor.

The VASP-16 is a multi processor system capable of performing rapid Q-measurements, FFT and Sliding-FFT analysis on beam-position pick-up data [3]. A characteristic of DSP systems, including the VASP-16, is the use of specially-written software routines to exploit the hardware of the system in order to achieve maximum performance. These special routines cannot be applied directly to another system architecture, and consequently some modification of the software is required if it is to be ported to another system.

An overview of the architecture of the VASP-16 will be given in Chapter 2, followed by an introduction to general programming of the system in Chapter 3. This provides a reference document for system maintenance.

In Chapter 4, the software written for the PS Q-measurement system will be explained with comments on its portability to other DSP systems, including a breakdown of the Q-measurement, the FFT and the Sliding-FFT functions. The specialised routines of the VASP-16 will be highlighted, allowing the PS Q-measurement software to be re-used with the equivalent routines of other systems replacing those of the VASP-16.

Contained in the appendices are the default hardware conditions and complete software listings for the PS Q-measurement system.

2. VASP-16 ARCHITECTURE

The VASP-16 is a multi-processor DSP card aimed at image-processing applications [4]. Running originally under OS9, a new device-driver has been written for the LynxOS operating system [5].

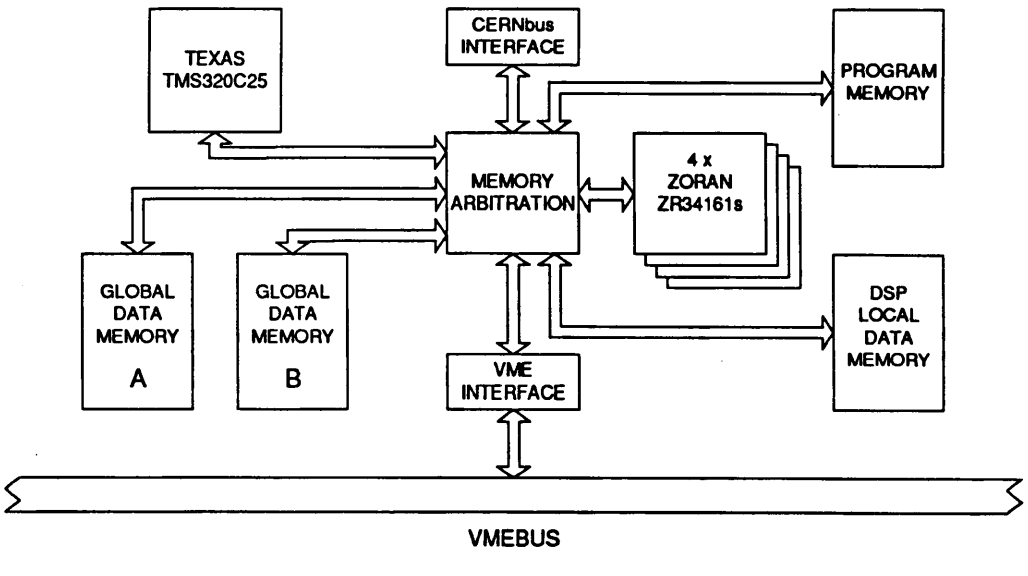


Fig. 1. Block diagram showing the internal architecture of the VASP-16.

The card contains five processors : a master Texas TMS320C25 and four slave Zoran ZR34161 Vector Signal Processors (VSPs) (see Fig. 1). The TMS320C25 is programmed in C and forms the heart of the system, handling VASP-16/VME communication, acquisitions, data management and function calls to the Zoran VSPs : specialised 16-bit processors with an architecture optimised for rapid DSP operations such as FFTs, convolution, correlation, etc. [6].

Zoran VSPs may perform arithmetic operations with three levels of precision : *integer, fixed floating-point* and *block floating-point.* The data which is transferred into the Zorans for processing, e.g. acquisition samples, must be in integer format. FFT analysis of the data uses block floating-point precision which gives better performance than fixed floating-point calculations. The spectrum calculated by the Zorans must however be stored in VASP-16 global memory in integer format.

Transfer of data into the VASP-16 global data memory is over a 16-bit open-architecture bus, MAXbus, which has been customised and renamed CERNbus. Global memory consists of two, 32 kbyte (16-bit) data-banks sharing the same address space, access to them being governed by a programmable arbiter. This architecture allows concurrent acquisition in one bank while processing the contents of the other and consequently VASP-16 measurements may be interleaved to obtain a greater measurement rate.

3. GENERAL VASP-16 PROGRAMMING

3.1. VME/VASP-16 communication window

Communication between the VME host and the VASP-16 is performed through a *communication window.* This window is an area of VASP-16 memory which is directly mapped into VME address space, and so any data which exists within the window can be read or altered by both the VASP-16 and the VME host.

The size of the window is fixed at 32 kbytes in length (16-bit bytes). The window starts at VASP-16 address `0x8000` and continues upwards until `0xFFFF`. This memory space corresponds to VASP-16 global memory, i.e. the two 32 kbyte data-banks which can be accessed by the VME host, the TMS320C25 and the four ZR34161s. As well as for data transfer between the VASP-16 and VME, it is used for storing acquisition data from CERNbus and the results of FFT calculations.

3.2. Global memory section

The two global data-banks share the same address space, with access being governed by a programmable arbiter (see Fig. 2 overleaf). Addresses below `0x8000` represent local memory to which only the TMS320C25 has access.

When allocating global variables, i.e. variables which can be accessed by all of the various processors, there is no differentiation between the two banks. When a vector `x[]` is declared, <u>it's space exists in both banks</u>. The user must use the arbiter to access `x[]` in either bank A or B. Software example 1 overleaf illustrates allocation of this memory using assembler language (asm) statements. All variables which must reside in global memory are declared in this way.

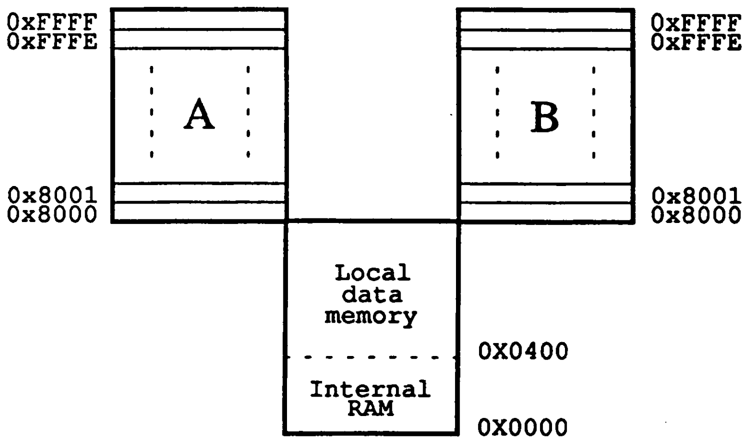


Fig. 2. Location of global memory data-banks A and B.

Software example 1

```
#include <vasp16.h>     /* always! */

asm("_rawdata      .usect    .global, 2000h");
asm("_fftresult    .usect    .global, 512h");
asm("_myflag       .usect    .global, 1h");
asm("              .globl    _rawdata, _fftresult, _myflag");
extern int rawdata[], fftresult[], myflag;
```

The first vector, `rawdata[]`, is located at address `0x8000`. `fftresult[]` is then automatically located at the next free address upwards, `0xA000`, and so on. The variable `myflag` is simply a 16-bit integer, located at address `0xA512`.

3.3. Data-bank arbitration

The Q-measurement system contains a number of components capable of requesting access to memory. The *data bank arbitration mode* determines which of these sources is allowed access at any particular time (see Fig. 3). Consisting of the Texas TMS320C25, the four Zoran ZR34161s, the VMEbus host master and CERNbus, these sources are governed by two modes: simple *single-bank* operation (mode 1) and more complex *dual-bank* operation

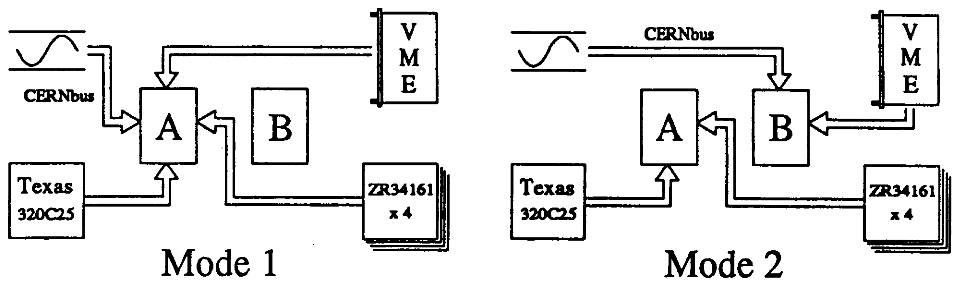


Fig. 3. Figure shows the two arbitration modes for access to global memory. In Mode 1, all processors have access to the same bank. In Mode 2, one bank can be processed while new data is stored in the other.

(mode 2).

In mode 1, all four sources may access only one data-bank at a time (Bank A in Fig. 3). Bank B is not redundant however: it can still be accessed by executing a *swap* function which changes access from bank A to B.

Mode 2 is more complex in that it splits access between the four sources. In Fig. 3 the TMS320C25 and the Zoran processors have access to bank A and can therefore process the data which is contained there. The CERNbus and VME host on the other hand have access to bank B. Here the possibility of concurrent processing and acquisition is introduced. While the TMS320C25 proceeds with its calculations, CERNbus can be loading data into bank B. The swap function can then be used to allow the TMS320C25 access to the new data in bank B, while CERNbus starts putting more data into bank A.

Defining the *current data-bank* as that bank which is accessible from the TMS320C25, the code shown in software example 2 sets bank A as the current data-bank in arbitration mode one, thus granting access to the VMEbus, CERNbus, TMS320C25 and the Zorans (see Fig. 3). Using the function `swpbnk()`, bank B would become the current bank and thus be accessed.

Software example 2

```
setarb(1);
setbnk(A);      /* points A to TMS320C25 & to VMEbus/CERNbus */
```

In this next example, bank A is set as the current bank in arbitration mode 2:

Software example 3

```
setarb(2);
setbnk(A);      /* points A to TMS320C25, B to VMEbus/CERNbus */
```

Accesses to bank A can now only be performed by the TMS320C25 and the Zorans, while bank B is available for VME or CERNbus transfers. The function `swpbnk()` would toggle the access between the two banks.

3.4. VME/VASP-16 communication

VASP-16 global memory is directly mapped into VME memory and hence it is very easy to transfer data from the VME host into the VASP-16 and vice-versa. Software example 4 overleaf shows all that is necessary to perform synchronisation between the two systems and subsequently transfer data.

The VME host firstly initialises the VASP-16 by calling the function `v16Open()`. This function returns a pointer to the board descriptor structure, `b0`, and its value is used in all future calls to the VASP-16. After initialisation, the function `v16Run()` is used by the VME host to download and run the VASP-16 module *testprog*. The host then calls `v16BoardF(b0)` which waits for a flag from the VASP-16.

Software example 4

VASP16 Communication code :-

```
/* testprog.c */

#include <vasp16.h>     /* Always! */
#include <qmeas.h>      /* user file */

main()
{
 int      *pcmd;

 setarb(1);
 setbnk(A);
 *pcmd = (int)(&x[0] - VMEOFFSET);
 flagt();                              /* Flag VME host          */
 dputd(pcmd, 1);                       /* Give host address      */
 hostf();                              /* Wait on flag from host */
 v16_exit();                           /* End                    */
}
```

VME Host Communication code :-

```
#include <v16Def.h>     /* Always! */
#include <psqmeas.h>    /* user file */

main()
{
 V16_DESC          *b0, *v16Open();
 int               dummy = 0x1234;
 unsigned short    *logical_base = 0x80000000, addr;

 b0 = v16Open(BASE, IVEC, VERBOSE, ABORT);
 v16Run(b0, "testprog");                    /* Run VASP16 module */
 v16BoardF(b0);                             /* Wait for flag     */
 addr = (unsigned short *)(v16GetD(b0));    /* Get address       */
 *(logical_base + addr) = dummy;            /* Write word        */
 v16FlagT(b0);                              /* Flag the VASP     */
 v16Close(b0);                              /* Close the board   */
}
```

The downloaded VASP-16 program `testprog`, which has been set running automatically, performs two important tasks without which communication is impossible. It firstly sets the arbitration mode to single bank before selecting global data-bank A as the current bank. In the subsequent program line, the VASP-16 calculates a pointer `*pcmd` which is used to give the host the address of a vector in global memory called `x[]`, this variable having been defined in the file `qmeas.h`. This form of pointer address calculation is always used; vectors with different names should have their name inserted instead of `x[]`, e.g. `*pcmd = (int)(&rawdata[0x400] - VMEOFFSET);`.

Having calculated a pointer to `x[]`, its value is transferred to the waiting host in a two-step operation. The host is firstly given a flag, using `flagt()`, to indicate that a transfer is following. The function `dputd()` is then called which writes the contents of the pointer to a data port. Meanwhile, upon receipt of the flag, the host has called `v16GetD()` in order to read the data port. The VASP-16 waits until the data port has been read and the two programs then continue. The final pair of flags between the VASP-16 and the host, `hostf()` and `v16FlagT(b0)`, are simply used as synchronisation before closing both programs.

3.5. VME interrupts

A VME interrupt can be sourced from the VASP-16 using the `user()` function. Software example 5 shows the code used by the VME process to wait upon this interrupt.

Software example 5

VASP16 code for VME interrupts

```
#include <vasp16.h>

main()
{
 int dummy = 4;    /* User defined value between 0-255 */

 /* Send an interrupt to the host */

 user(dummy);

 /* etc */
}
```

VME code to await a user interrupt

```
#include <v16Def.h>      /* Always! */
#include <psqmeas.h>     /* user file */

main()
{
 V16_DESC *b_Desc;

 /* etc */

 /* Wait on an interrupt from the host while sleeping */

 b_Desc->Status = V_USER;
 read(b_Desc->Des, V_USER, (char *)0);

 /* etc */
}
```

3.6. CERNbus acquisition transfers

A 16-bit open-architecture bus is available on the VASP-16 for transferring data both into and out of the board. This has been modified for the PS application and will therefore be referred to as CERNbus. Programming CERNbus is different from programming the MAXbus, and in this respect the reader should ignore the manufacturers documentation and instead use the following examples.

CERNbus is used to transfer data into the VASP-16 in the form of 16-bit bytes. To illustrate its use, the following example is now explained.

Software example 6

```
#include <vasp16.h>
#define  SING      0x8      /* Single Measurement Mode */

asm("_data1   .usect   .global, 1000h");
asm("_data2   .usect   .global, 1000h");
asm("_myflag  .globl _data1, _data2, _myflag");
extern int    data1[], data2[], myflag;

void bankbint()
{
 myflag = 1;
}

main()
{
 setarb(1);
 setbnk(B);                     /* Bank B -> CERNbus & TMS320C25    */
 roi(0, MASTER);                /* Enable tri-state MODE signal     */
 setcern(SING);
 myflag = 0;                    /* Set flag to acq-in-progress      */
 insmax(bankbint);              /* Install CERNbus interrupt        */
 mbase(data1 - 0x8000);         /* Give address of data1            */
 getcern(~(512) + 1);           /* Acquire 512 points in data1      */
 while (myflag == 0) {}         /* Wait for end of acquisition      */
 myflag = 0;                    /* Repeat for data2,but 256 points */
 mbase(data2 - 0x8000);
 getcern(~(256) + 1);
 while (myflag == 0) {}         /* wait                             */
 dinmax();                      /* De-install CERNbus interrupt     */
 /* etc */
```

As explained in section 3.1., asm statements are used to define the global memory allocation. `data1[]` is a vector of size `0x1000` bytes, located at address `0x8000`, and `data2[]` is of the same length, but located at address `0x9000`. With global memory thus defined, the main program can now be examined.

After setting bank B as the current bank with `setbnk(B)`, under arbitration mode 1, the program calls two functions which define the *acquisition mode* of CERNbus. The first function, `roi(0, MASTER)`, is used to enable a tri-state signal called `MODE` which is needed by the external interface containing the ADC. The second function, `setcern()`, defines the state of the `MODE` signal. In the example, `setcern(SING)` programs the CERNbus into *single measurement* mode. This doesn't mean that only one measurement will be made, but means that only measurements shall be made which can be accommodated within one data-bank, i.e. acquisitions of less than `0x8000` bytes in length. Acquisitions larger than this can be performed when the CERNbus is set into the second mode. This mode shall be explained shortly with a second example.

When an acquisition is programmed, the TMS320C25 must wait until the end of the acquisition before trying to process the data. The way in which this has been achieved does not correspond to the manufacturers documentation. The manufacturers functions rely upon the contents of registers and register-images. Failure to correctly update these registers by the VASP-16 results in hang-ups and incorrect acquisition modes. The reader must use the form presented in this example if a program is to work correctly.

In the example, a flag called `myflag` is declared in global memory. Before performing the acquisition this flag is set to `0`. A simple interrupt, `bankbint()`, is then installed by calling the function `insmax()`. The TMS320C25 can now proceed as normal, with the interrupt active in the background. Upon receipt of an end-of-acquisition signal, the interrupt is executed. Hence, as in the example, the TMS320C25 can loop, testing the value of `myflag`, until it is set by the interrupt thus indicating a completed acquisition.

The acquisition itself is called by two simple functions, `mbase()` and `getcern()`. The first of these is used to inform CERNbus where the data is to be stored. In the example, `mbase(data1 - 0x8000)` passes the address of the vector `data1` relative to the base of global memory, i.e. `0x0`. The use of `mbase(0x0)` would have produced the same result, but using vector names allows an easier appreciation of where data is to be found. Once set, `mbase()` does not need to be called for subsequent acquisitions unless the data is to be stored in a different location. To finally arm CERNbus, the function `getcern()` is called, which leaves CERNbus awaiting an external trigger, ready to acquire 512 samples. Note that the two's complement of the amount of data to be acquired is passed to this function.

All acquisitions which are smaller than 32 kbytes can be made in the described manner. For larger acquisitions the continuous measurement mode must be used. Again, an example will be used to illustrate the method.

In software example 7, a single area of memory, `data[]`, has been set aside for data. This vector is exactly 32 kbytes in size, corresponding to a complete global data-bank. It is the aim of the program however to perform a 64 kbyte acquisition. This requires the use of both data-banks and so *arbitration mode 2* and the *continuous measurement mode* are selected. Once done, an interrupt is again installed, not this time to only detect the end of the acquisition but also to initiate the second 32 kbyte acquisition. This interrupt, `contint()`, calls the assembly function `swaparm()`. The purpose of `swaparm()` is to swap access to the data banks, i.e. giving CERNbus access to the remaining empty bank, and then re-arm CERNbus ready to complete the acquisition.

Software example 7

```
#include <vasp16.h>
#define  CONT      0x0       /* Continuous Measurement Mode      */

asm("_data    .usect    .global, 8000h");
asm("_myflag  .globl _data,_myflag");
extern int    data[], myflag;

void contint()
{
 swaparm(0x0, (~(0x8000) + 1));
 dinmax();
}

main()
{
 setarb(2);                   /* Set dual-bank arbitration       */
```

```
setbnk(A);                  /* A -> TMS; B -> CERNbus           */
mbase(0x0);                 /* Set base address for data        */
roi(0, MASTER);             /* Enable tri-state MODE signal     */
setcern(CONT);              /* Continuous measurement mode      */
insmax(contint);            /* Install interrupt routine        */
getcern(~(0x8000) + 1);     /* Acquire first 32k in bank B      */
mwait();

/* Second acquision completed here   */
/* Now process...                    */
```

The first 32 kbytes of the acquisition are obtained by calling `getcern(~(0x8000+1)`. Upon receipt of an external trigger, this data is stored in bank B. When the end-of-acquisition signal is received, the interrupt `contint()` executes, swapping the banks and re-arming CERNbus. Since only one external trigger is received, a false trigger is generated by the ADC interface, the delay between the end of the first acquisition and the new trigger being defined by hardware. The final function in the main program, `mwait()`, is used to detect the completion of the second acquisition. When returned from this call, the TMS320C25 can then proceed and process the data.

3.7. FAST FOURIER TRANSFORMS

3.7.1. Introduction

The principle task of the VASP-16, to perform fast signal processing, is given to the four Zoran ZR34161 VSP processors. A library of functions is available, containing operations such as vector addition and multiplication, window generation (e.g. Blackman, Hamming), rectangular to polar conversion, and both real and complex FFT calculation using either fixed or block floating-point arithmetic. The FFT can be computed for input data lengths of 256, 512 or 1024 samples.

The different FFTs in the library require certain vectors to be declared in VASP-16 global memory. The lengths of these vectors change depending upon the size of the FFT, and exact details can be found in the manufacturers documentation.

All input data (acquisition samples) which is to processed is stored in the vector `x[]`, at location `0x0`. Input values are always in integer format, and this is also true of the resulting spectrum, which is located in `y[]`. The other vectors are used by the FFT itself and are not of concern to the user. The allocation example shown below is that of the 512-point block floating-point FFT with magnitude-squared output.

Software example 8

```
#include <vasp16.h>

asm("_x       .usect   .global, 512");
asm("_y       .usect   .global, 512");
asm("_t       .usect   .global, 1024");
asm("_sect2   .usect   .global, 1");
asm("_max     .usect   .global, 8");
asm("_scale   .usect   .global, 2");
asm("        .globl _x, _y, _t, _sect2, _max, _scale");
extern int   x[], y[], t[], sect2, max[], scale[];
```

3.7.2. Pre-scaling of input data

During the FFT calculation a rather complicated automatic-scaling mechanism is used to prevent the amplitudes of the lines in the spectrum from increasing beyond the range of the 16-bit processors. Rather than complicate the picture with an explanation of this process, a description will now be given of a new method which has been developed in order to eliminate occasional arithmetic-overflow errors which were found during the development of the PS Q-measurement system.

When called, the FFT function has an integer value passed to it which controls the amount of scaling performed during the FFT process. This value, `minscale`, must be selected by performing some pre-processing on the sampled input data. The value of `minscale` must lie within `0` and `9`, and is dependent upon the approximate magnitude of the data within the sample and the size of the FFT itself. The value must be chosen such that the return value of the FFT function (known as the binary exponent) is zero. A binary exponent greater in value than zero indicates that a larger `minscale` value should be used.

Fig. 4 shows the effect of the `minscale` parameter upon the amplitude of the FFT spectrum. If a low value is used, e.g. 0-3, then a small amplitude input will be *magnified* and have good spectral amplitude resolution. Alternatively, a large amplitude input signal, processed by an FFT with a low `minscale` value, could produce overflow errors (the peak in the spectrum is lost and a value of zero appears).

At the other end of the `minscale` range, large `minscale` values should be selected for large amplitude sampled input data. A trial and error method is required in order to discover the `minscale` value which best suits a particular input sample. It is recommended that a value be selected such that the resultant spectral amplitude does not exceed `12000` (decimal) at any point in the spectrum, with a binary exponent of zero returned from the FFT.

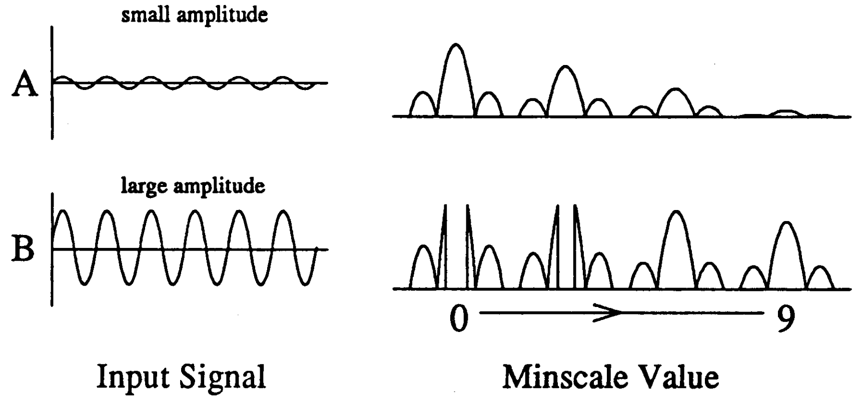


Fig. 4. The effect of the parameter `minscale`. (a) For a small amplitude sample, a low `minscale` value gives good spectral amplitude resolution. A large `minscale` value reduces the spectral amplitude however. (b) For a large amplitude sample, a low `minscale` value gives too much spectral amplitude resolution, and the spectral peak can overflow. A larger value of `minscale` must be used to reduce the spectral amplitude.

3.7.3. FFT execution

Assuming that an acquisition has been performed and the data stored in the vector `x[]`, the following code is used to perform the 512-point magnitude-squared block floating-point FFT:

Software example 9

```
sect2 = stvmd(SECT_2 + ILI_ON + FAST);
bexp = fftm512(minscale);
```

The first line is used to run the ZR34161s in their fastest mode and is only called once, while the second line executes the FFT itself. The `minscale` parameter passed to the FFT function has already been explained in section 3.7.2, and the parameter `bexp` is the binary exponent which must have a return value of zero. The two other FFT functions, for 256 and 1024 points, are called using `fftm256()` and `fftm1024()` respectively. Table 1 shows the execution times of the VASP-16 FFT functions.

Table 1. VASP-16 FFT execution times.

FFT function	Execution time (ms)
fftm256()	0.437
fftm512()	0.628
fftm1024()	1.110

There exists a problem with the FFTs which has been solved. If, using the example presented, an FFT is continually executed upon the same data, the resultant spectra will vary in amplitude periodically. This error is overcome by running the code given in software example 10 **before every FFT.**

The four lines of code shown reset the internal FFT scaling pointers which are contained within registers of the Zoran processors themselves.

Software example 10

```
setvmd(1, RSS);
setvmd(2, RSS);
setvmd(3, RSS);
setvmd(4, RSS);
```

3.7.4. Re-scaling of spectra

The amplitude resolution of the FFT spectrum is controlled by the value of the `minscale` parameter. To obtain the true amplitude (A_n) of the spectrum, the amplitude of every spectral line (k_n) must be multiplied by a re-scaling value. This value is a function of the FFT size and `minscale`. Listed below are the re-scaling formulae for the 256, 512 and 1024-point block floating-point magnitude-squared FFTs.

256-point FFT $A_n = \sqrt{k_n} \times 2^{(\text{minscale} + 1)}$ $n = 1,2,...,256$ (3.1)

512-point FFT $A_n = \sqrt{k_n} \times 2^{(\text{minscale})}$ $n = 1,2,...,512$ (3.2)

1024-point FFT $A_n = \sqrt{k_n} \times 2^{(\text{minscale} - 1)}$ $n = 1,2,...,1024$ (3.3)

These formulae may only be applied to spectra calculated using an FFT which has returned a binary exponent of zero.

4. PS Q-MEASUREMENT SOFTWARE

4.1. Porting the software to other DSP systems

DSP programming invariably involves writing software which exploits the architecture of the system in order to achieve maximum processing performance. The VASP-16 uses a combination of "C", assembler code and a specialised vector-oriented language used by the Zoran processors.

The Q-measurement software is mainly written in "C", but makes a large number of function calls to routines which are specific to the VASP-16, e.g. data-acquisition routines, FFT calculations, etc. In this chapter, the purpose of these functions will be identified so that they can be replaced with the equivalent routines of other DSP systems. In the following example these functions are highlighted using a bold, italicised font. This style shall be used throughout the remainder of the chapter.

Software example 11

```
insmax(aint);                            /* VASP-16 routine */
myflag = 0;                              /* standard C */
setcern(SING);                           /* VASP-16 routine */
mbase(x - 0x8000);                       /* VASP-16 routine */
getcern(~(localcmd[0].fftsize) + 1);     /* VASP-16 routine */
while (myflag == 0) {}                   /* standard C */
```

4.2. Overview of the hardware

Fig. 5 shows the hardware of the PS Q-measurement system. Acquisitions are made using a 12-bit Analogue to Digital Converter (ADC) which has a maximum sampling rate of 10 MHz. Prior to sampling, the pick-up signal must be passed through a bandpass anti-aliasing filter because the frequency content of the beam extends well beyond the sampling frequency.

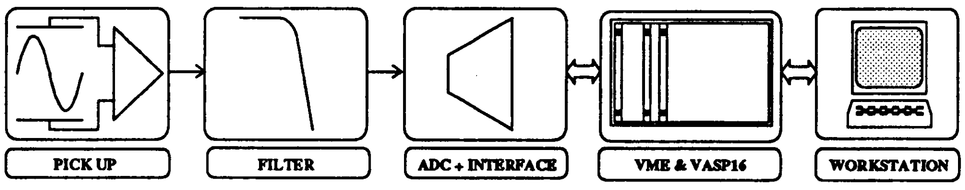


Fig. 5. PS Q-measurement system hardware.

The ADC is currently located on a separate NIM chassis, and transfer of data into the VASP-16 is over a 16-bit open-architecture bus, CERNbus.

The VME chassis is a MVME147 diskless system, running on LynxOS. The final link, between the VME chassis and the control workstation, is currently under development and uses the PS/CO control protocol. In the meantime, a local display has been developed which allows plotting of Q-measurements and Sliding FFT displays.

4.3. Introduction to the software

The Q-measurement software is designed to run in a continuous loop, corresponding to the PS machine cycle. The loop commences when the VASP-16 receives a list of instructions containing the total number of measurements to be performed, their types, FFT sizes, etc. (see Fig. 6).

The program proceeds by checking if the measurement type is the Sliding FFT. This is an exclusive measurement which may not be performed in the same cycle as other measurement types because of the very large volume of data it produces. If selected, the main program loop will repeat after completion of the Sliding FFT calculations and alternatively, if not requested, the program will execute the list of measurements sequentially using concurrent data-acquisition and processing.

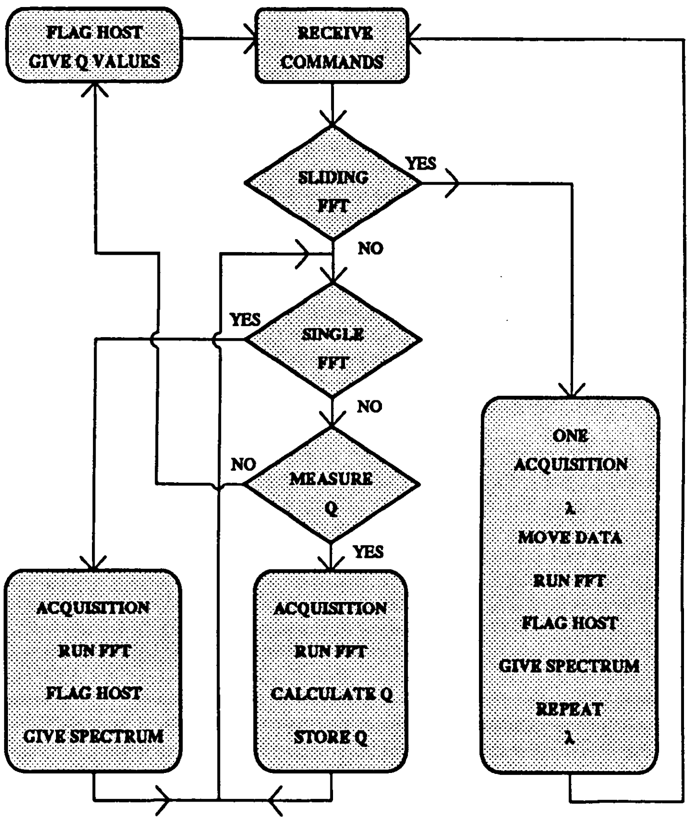


Fig. 6. VASP-16 program simplified view.

The Sliding FFT is used to analyse 52 kbytes of data with 256, 512 or 1024-point magnitude-squared FFTs. A 26-kbyte acquisition is initiated by an external trigger and is stored in data bank A. Upon completion, the program grants access to data bank B and a false trigger, generated by the ADC interface, initiates the second 26 kbyte acquisition, with synchronisation to the same bunch. The number of FFTs specified in the measurement list is then executed and the spectra are returned to the host upon completion of each. The user may ask for the Sliding FFT raw-data and not the spectra if desired. In this case, a 64-kbyte acquisition is performed.

Between FFT calculations, the FFT window is moved along the data by a number of samples as specified by the user. The resultant spectra therefore represent the beam oscillation in frequency and time. A small step can be used to obtain a very detailed picture of the beginning of the acquisition or alternatively a large step can be used to obtain a general view of the evolution throughout the whole acquisition.

Q-measurements and FFTs are interleaved, using background interrupt routines, to give concurrent acquisition and processing. An acquisition is armed for the first measurement in the list and is initiated by an external trigger. Upon completion, the program looks for the subsequent measurement, if any, and arms the next acquisition which may occur concurrently with processing of the first. This continues until the list of measurements is exhausted and the loop is then repeated.

The FFT is the simplest measurement and may contain 512 or 1024-points, magnitude squared. The number of samples acquired, N, equals the number of points in the FFT. This data is pre-processed to prevent arithmetic overflows during the many cross-multiplications of the FFT. Upon completion of the FFT calculation, the spectrum, containing N/2 points, is returned to the VME host along with some scaling information. This information is used to re-scale the magnitude-squared spectrum to give the true amplitude spectrum. Alternatively, the user may ask for the raw acquisition data instead of the spectrum.

The Q-measurement makes use of the 512-point FFT only, and the *q*-value is calculated by interpolating the peak in the spectrum. The reader is referred elsewhere for more information on the theory of the q-calculation [7].

4.4. Global memory section

Listed below is an extract from the header file `qmeas.h`. This code defines the allocation of VASP-16 global memory which is used for storing acquisition data, FFT results, etc. For the VASP-16, a global variable is one that can be accessed by all the on-board processors, CERNbus etc. Other DSP systems will have something similar.

Software example 12

```
asm("_x           .usect    .global, 6c00h");
asm("_y           .usect    .global, 800h");
asm("_t           .usect    .global, 800h");
asm("_sect2       .usect    .global, 1h");
asm("_max         .usect    .global, 8h");
asm("_scale       .usect    .global, 4h");
asm("_number      .usect    .global, 1h");
asm("_globalcmd   .usect    .global, 258h");
asm("_globalres   .usect    .global,  6h");
asm("_myflag      .usect    .global,  1h");
extern int    x[], y[], t[], sect2, number[];
extern int    max[], scale[], myflag;
```

The vectors `x[]`, `y[]`, `t[]`, `sect2`, `max[]` and `scale[]` are all dedicated to the FFT function. The declaration of such vectors is necessary for the VASP16 because arithmetic addressing is not possible with the Zoran ZR34161s: i.e. all variables must reside in pre-declared locations. Fortunately, the user is only concerned with `x[]`, in which acquisition data is placed, and `y[]`, which contains the spectrum calculated by the FFT. For individual

FFTs and Q-measurements, acquisition data is located at 0x8000, corresponding to x[0]. The location of Sliding FFT data is different and will be explained in section 4.7.1.

The integer value contained in number is the total number of measurements to be performed by the VASP-16. This value is set by the VME host for each cycle in which measurements are to be made.

4.5. Transfer of the measurement list

Before receiving the measurement list from the VME-host, the VASP-16 sets its operating parameters using the following code:

Software example 13

```
setarb(2);
setbnk(A);
roi(0, MASTER);
hostf();
*pcmd = (int)(&globalcmd[0] - VMEOFFSET);
flagt();                /*                                */
dputd(pcmd, 1);         /* Synchronise & list transfer    */
hostf();                /*                                */
```

It is important to understand the implications which this has. By selecting arbitration mode 2, the master Texas processor can only access one of its two data-banks at a time. Since both data-banks are to be used during operation, where should the measurement-list be stored so that the Texas can have access to it at any time without the need for time-consuming bank-swapping? The answer is simply to copy the command list into local memory. Hence it no longer matters which bank the Texas has access to because the command list is now a local variable.

The VASP-16 receives the actual measurement list after synchronising with the host. The measurement list is actually a global structure, and software example 14 explains the meaning of each field within it.

Software example 14

```
struct cmdlist {
     int identity;      /* measurement identification number    */
     int kind;          /* q measurement, fft or sliding fft    */
     int fftsize;       /* 256, 512 or 1024 point fft choice    */
     int quantity;      /* number of ffts (sliding fft only)    */
     int slidstep;      /* step-interval  (sliding fft only)    */
} extern globalcmd[];
```

Other DSP systems will of course use their own functions for host/system synchronisation, but the same list structure should be used if the software contained in Appendix A.2 is to run.

4.6. FFT analysis

4.6.1. Pre-scaling of input data

As discussed in section 3.7.2., data which is to be analysed using the FFT must be pre-processed to select an optimum value for the parameter `minscale`. The function `bestmscl()`, contained in Appendix A.2, is called from the main program and returns an integer value which is used as the `minscale` value in the FFT function. The function `bestmscl()` requires the main program to pass the FFT size because the choice of `minscale` differs for the 256, 512 and 1024-point FFTs.

Other DSP systems will probably not require any pre-processing of data before performing FFT analysis. Todays systems are now using typically 64-bit floating-point arithmetic, thus giving a much improved dynamic range and reducing the risk of overflow to a negligible level.

4.6.2. FFT execution

The FFT is executed by calling the function `fft(size)`, where `size = 256, 512` or `1024`-points. This function calls `zclear()` and `bestmscl()`, before using a switch statement to call the relevant FFT function itself. The value of minscale is returned to the calling program for use in re-scaling the spectra to their true amplitudes. The function `fft(size)` can be found in Appendix A.2. This function can be re-used in other DSP system by replacing the VASP-16 FFT function calls with those of the other system.

4.6.3. Re-scaling of spectra

To obtain the true amplitudes of spectral lines, the amplitude of each line must be multiplied by one of the functions (3.1) to (3.3) as explained in section 3.7.4. This is not performed by the VASP-16 because of the emphasis on fast processing, and so the following code is suggested to the reader to calculate the scaling value.

Software example 14

```
double    backscale;
int       minscale, size_of_fft;

switch (size_of_fft) {
     case 256:      backscale=ldexp(scale1, minscale + 1);
                    break;

     case 512:      backscale=ldexp(scale1, minscale);
                    break;

     case 1024:     backscale=ldexp(scale1, minscale - 1);
                    break;

     default:       break;
}
```

4.7. The Sliding FFT

4.7.1. Data acquisition

After receiving the command list from the host, the program firstly checks whether or not the Sliding FFT is to be performed. As mentioned previously, acquisition data is stored in the vector `x[]`. However, a straightforward acquisition cannot be performed. If an acquisition were to be performed from `x[0]` upwards, data would be corrupted as soon as new data was moved into the FFT window. Fig 7 illustrates the scheme which has been used to overcome this effect.

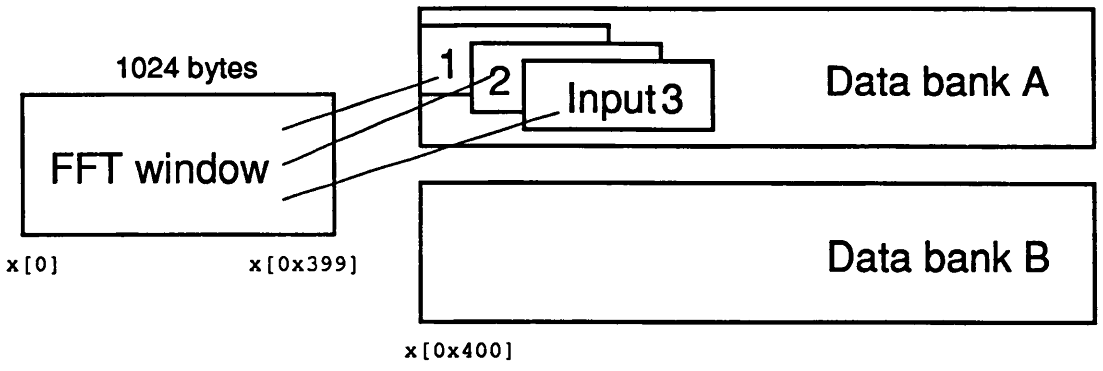


Fig. 7. Sliding FFT operation. Data is stored from `x[400]` in both banks, and copied in blocks into the FFT window for analysis, thus preventing fragmentation of the data.

The FFT input window is left free for loading data. The acquisition then makes use of the space in both data banks from `x[0x400]` upwards. This now allows data to be copied into the FFT window when the FFT is slid along the acquisition.

The disadvantage of slightly less space for acquisition data is outweighed by the simpler routine for performing the data manipulation. How much data then is available for processing? `x[]` is of size `0x6C00`. We take away `0x400`; thus we have left `0x6800`, i.e. 26624 bytes. We use both banks for the Sliding FFT however to give us a total of 53248 bytes, and at a sampling rate of 500 kHz, this represents an acquisition lasting for 106 ms.

Software example 15 shows the code used to perform the Sliding FFT acquisition. A background interrupt is installed which handles the end-of-acquisition event for the first data-bank. When half of the total acquisition is completed, this interrupt will re-arm CERNbus and the second acquisition will be performed, initiated by a false hardware trigger.

Software example 15

```
sfft_start = 0x400;
insmax(contint);        /* Install background interrupt routine */
mbase(sfft_start);      /* Set data-base addr. to x[0x400] */
setcern(CONT);          /* Set continuous acquisition mode */
getcern(FULLACQ);       /* First half of acquisition executed */
mwait();                /* Wait for completion of second half */
```

The function `mwait()` causes the main program to halt until the entire acquisition is completed.

Other DSP systems will probably not require the acquisition scheme illustrated in Fig 7. Arithmetic addressing is usually available, thus allowing the FFT to be truly stepped along the data. However, multiple bus systems introduce their own peculiarities and some strange manipulation may be necessary nevertheless.

4.7.2. FFT movement

Although the name *Sliding* FFT is used, it is not the FFT but rather the data which is moved. Simple copies of the data are made into the FFT window. A little manipulation is required when the required data lies within both banks. When each FFT has been performed the spectrum and the relevant scaling information is passed back to the host. The FFT operation is the same as that described in section 4.6, and is repeated the number of times specified in `localcmd[0].quantity`.

4.8. Q-measurement

The q-value is calculated using the function `qmeas()` which can be found in Appendix A.2. The flow diagram in Fig 8 shows the operation of this function.

As in the case of the FFT, the Zorans are firstly reset using `zclear()`. `bestmscl()` is then called in order to obtain an approximate value for the oscillation size and subsequently select the optimum minscale parameter used by the FFT. `qmeas()` uses the results of `bestmscl()` to decide whether or not to perform the q-calculation at all. If it finds that the input raw data magnitude is below the value of `QTHRESH`, no q-calculation is performed and a value of `q = 0.0` is returned to the host. This is programmed to prevent q-analysis based on low resolution spectra.

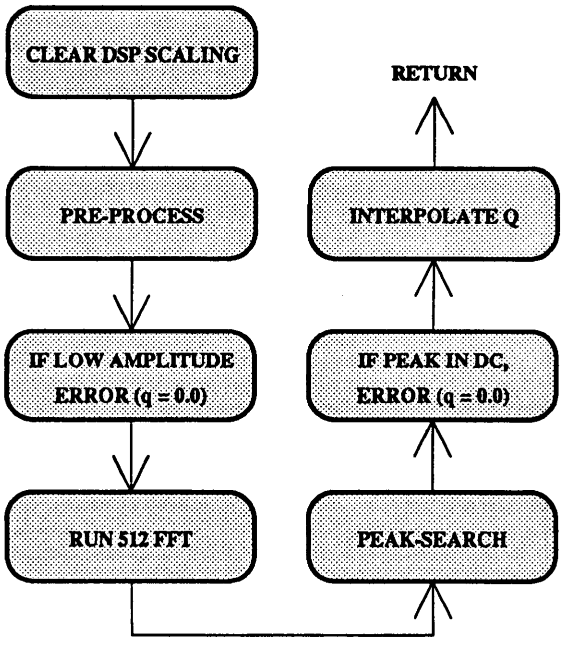


Fig. 8. Flow diagram showing the operation of the function `qmeas()`.

The FFT performed is the 512-point block floating-point, which takes 0.628ms to execute. Following FFT execution, a peak search is performed which scans the FFT spectrum from high frequency to low. The first 20 bins of the FFT (containing DC and low frequency components) are not searched to prevent confusion between the tune-peak and DC. A little analysis is then performed to discover the nature of the spectrum, e.g. if the second highest spectral line is to the left or right, etc. This is then used by a switch statement to execute interpolation of the actual q value.

This function can almost be used without modification in other DSP systems. All that needs to be changed is the removal of zclear(). It may not be necessary to perform pre-processing using bestmscl() in systems with high floating point resolution.

5. CONCLUSIONS

This paper presents a guide to general VASP-16 programming and the code developed for the PS Q-measurement. This should be useful to anyone required to work with the existing system, or develop a similar system on another type of DSP board. Although the software cannot be re-used without modification, some attempt has been made to show those parts of the code which may be replaced with equivalent functions of other systems, and also those parts of the code which are superfluous.

It will be the case, however, that DSP systems rely on very specific code in order to achieve maximum performance from the hardware, and the software will thus be limited in terms of portability.

6. ACKNOWLEDGEMENTS

I would like to thank the members of my section, Messrs. E. Schulte, J. Gonzalez and J. Belleman for their willingness to discuss the project throughout its development. Thanks also to N. de Metz Noblat who wrote the LynxOS device driver.

7. REFERENCES

[1] G.C. Schneider, "Q-measurement with swept RC filter", CERN/MPS/SR 73-4, 22.05.73.

[2] C. Kessler, "Betatron Tune Measurement in the PS using the FFT", CERN/PS/PA Note 90-27, 30.08.90.

[3] S. Johnston, "Performance of the PS FFT Q-measurement system", CERN/PS/BD/Note 92-8, November 1992.

[4] Future Digital Systems Ltd., "Host User Manual" and "DSP User Manual", 1989.

[5] LynxOS device driver written for the VASP-16 by Nicolas de Metz-Noblat, CERN PS Division, 18.03.92.

[6] Zoran Corporation, "Digital Signal Processors Data Book", 1987.

[7] E. Asseo, "Guide méthodique pour la mesure (valuers corrigées) des oscillations bétatroniques dans LEAR", PS/LEA Note 85-4, April 1985.

A. APPENDICES

A.1. Default hardware configuration for the PS

Chapter 3 of the VASP16 Host User Manual lists the configurations which can be set using hardware jumpers. The set-up given below ensures correct operation of the VASP-16 for the PS Q-measurement system. The link positions presented here assume that the reader is looking at the board with the Texas processor visible and the blue connectors at the 12 o'clock position. For link pack locations on the board please refer to the VASP16 Host User Manual, Chapter 3, "Configuration".

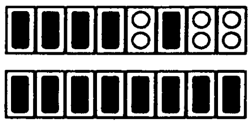
The pattern ○ represents an open link connection, while the pattern ▮ represents jumpers in-place.

Board Base Address

The VASP-16 global memory address `0x8000` is mapped into VME address space at the *board base address*. This value has been set using jumpers to `0x400000` and thus data contained at `0x8000` in the VASP16 can be read from VME physical address `0x400000`.

Location :

link packs H1 & H2.



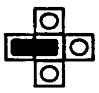
Bus Request Arrangement

Two sets of link packs are used to define the VME *Bus Request Level* and *Bus Grant Level*. The following jumper settings ensure both are set to level 3.

<u>Bus Request Level Selection</u>

Location :

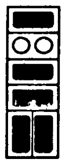
links LK1-5.



<u>Bus Grant Level Selection</u>

Location :

links LK6, 7, 10-19.



Since DMA is not being used, the BCLR (Bus Clear) signal is not required and accordingly this jumper on the H2 link pack is not inserted. The ACFAIL link has also been ignored, this being located in link pack H3.

Bus Busy (BBSY) Signal

To ensure that the VASP-16 may act as a VME bus master it is necessary to insert a jumper between links LK48 & 49.

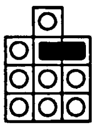
Interrupt Level Selection

This can easily be altered by the user - it is only necessary to ensure that both the interrupt output and acknowledge levels are the same. Interrupt level 5 has been set using the jumper positions shown below.

VME Interrupt output level selection

Location :

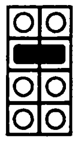
adjacent to J1 connector



VME Interrupt Acknowledge level selection

Location :

link pack H3



DMA Enable

Since DMA is not being used, the jumper between links LK8 & 9 should not be inserted.

VASP-16 reset from VME

It is important that this facility be enabled. A jumper should therefore be inserted between links LK20 & 21.

Mode Selection (TMS320C25)

The TMS320C25 is being operated in MicroProcessor mode and therefore the user should ensure that a jumper is **not** present between links LK1 & 16 of the H4 link pack.

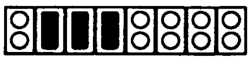
Device Interrupt Enable (TMS320C25)

Three interrupts are available to interrupt TMS320C25 operation from external devices and should be enabled as shown.

Location :

links 2-15,3-14,4-13 of

the H4 link pack



Auto-Monitor Entry (TMS320C25)

Automatic entry of the on-board monitor should be ensured by inserting a jumper between links LK38 & 39.

Bank-Swap Interrupt Enable (TMS320C25)

This function should be enabled by inserting a jumper between links LK42 & 43.

ECL Timing Termination (CERNbus)

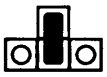
All link pairs of the link pack H7 should have jumpers inserted.

Primary/Alternate Port Select (CERNbus)

The CERNbus data output ports should be permanently enabled as shown.

Location :

links LK 33 & 35



XILINX

Jumpers in links LK30 & 59 and links LK56 & 55 are the default factory settings and should not be altered.

Local Data Memory (TMS320C25)

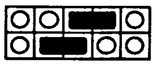
Two 32kx8 SRAM devices are used for this purpose requiring a jumper between links LK63 & 64.

Global Data Memory (TMS320C25)

Four 32kx8 SRAM devices are used to provide two 32k (16-bit) data-banks. Jumpers should be placed between links LK65 & 66 and links LK67 & 68.

Program Memory (TMS320C25)

This setup depends upon the type and size of memory devices being used. For the 8921M chips currently being used, the following jumpers are required.

Location :	**Setting :**
links LK83-86	no jumpers present
links LK69-72	jumper between LK69 & 72
links LK73-76	jumper between LK73 & 76
link pack H6	

Wait State Selection (TMS320C25)

No jumpers should be inserted between links LK51-54 to ensure zero wait-state operation.

A.2. VASP-16 SOFTWARE LISTING

In the following code, specific VASP-16 functions have been highlighted in a bold, italicised font. These functions, or combinations of them, should be replaced with equivalents of other systems if the software is being ported.

```
/*---------------------------------------------------------------------*/
/* psvasp.c                                                            */
/*                                                                     */
/* VASP16 module for the PS Q-measurement & FFT analysis               */
/*---------------------------------------------------------------------*/
/* Module is downloaded and executed by the main() thread of BODY.    */
/* It runs in a continuous loop, and performs the following:          */
/*                                                                     */
/* LOOP {                                                              */
/*       1. Receives measurement instructions from the DSC.           */
/*       2. If asked for Sliding FFT, it makes 1 large acquisition,   */
/*          then calculates the individual FFTs and passes them to    */
/*          the host after each calculation, who is waiting on an     */
/*          interrupt (function user() sources the interrupt)         */
/*       3. If Q or single-shot FFTs, runs a series of acquisitions   */
/*          and calculations.  Q values are passed upon completion    */
/*          of the whole number, whereas FFTs are passed back after   */
/*          calculation of each.  Interrupts are used to waken DSC.   */
/* } END_OF_LOOP                                                       */
/*---------------------------------------------------------------------*/
/* Created : September 1992                   S. Johnston, PS/BD.     */
/* Version : 1.0                                                       */
/*---------------------------------------------------------------------*/
/* Revised : 04/05/93                         S. Johnston, PS/BD.     */
/* Version : 1.0                                                       */
/* Comment : Interrupts introduced.                                   */
/*---------------------------------------------------------------------*/
/* Revised : 18/06/93                         S. Johnston, PS/BD.     */
/* Version : 1.1                                                       */
/* Comment :                                                           */
/*                                                                     */
/* Also now allows raw single-shot acquisitions and 64k max size.     */
/* qmeas() will ignore the first twenty bins, thus dc offset is not   */
/* a problem.                                                          */
/*---------------------------------------------------------------------*/

#include <vasp16.h>       /* Future Digital Systems stuff */
#include <psvasp.h>       /* My declarations */
#include <math.h>



void contint()
{
 /* --- Interrupt routine for CERNbus ------------------------------*/
 /*                                                                 */
 /* Executed upon the end of the first half of the acquisition for */
 /* Sliding FFT measurements.  This swaps the banks and re-arms    */
 /* CERNbus for a second 0x6800 samples acquisition.               */
 /*-----------------------------------------------------------------*/

 swaparm(DATALOC, FULLACQ);
 dinmax();                    /* Un-install interrupt routine */
 return;
}
```

```
void fullint()
{
 /* --- Interrupt routine for CERNbus -------------------------------*/
 /*                                                                  */
 /* Executed upon the end of the first half of the acquisition for  */
 /* Sliding FFT measurements.  This swaps the banks and re-arms      */
 /* CERNbus for a second 0x8000 samples acquisition.                 */
 /*------------------------------------------------------------------*/

 swaparm(0x0, ALLBANK);
 dinmax();                    /* Un-install interrupt routine */
 return;
}


void aint()
{
 /* --- Interrupt routine for CERNbus (Bank A) ---------------------*/
 /*                                                                 */
 /* Upon the end of an acquisition, this function sets myflag. This */
 /* flag is being tested by main....a form of semaphore if you like */
 /*-----------------------------------------------------------------*/

 myflag = 5;
 dinmax();                    /* Un-install interrupt routine */
 return;
}


void bint()
{
 /* --- Interrupt routine for CERNbus (Bank B) ---------------------*/
 /*                                                                 */
 /* Upon the end of an acquisition, this function sets myflag. This */
 /* flag is being tested by main....a form of semaphore if you like */
 /*-----------------------------------------------------------------*/

 myflag = 2;
 dinmax();                    /* Un-install interrupt routine */
 return;
}


void zclear()
{
 /* --- Clear Zoran VSP scaling pointers ---------------------------*/
 /*                                                                 */
 /* This function is called before EVERY FFT calculation.  If not   */
 /* used, the spectra may not be correct!!!                         */
 /*-----------------------------------------------------------------*/

 setvmd(1, RSS);
 setvmd(2, RSS);
 setvmd(3, RSS);
 setvmd(4, RSS);
 return;
}
```

```
int bestmscl(size, acqdata)
int size, *acqdata;
{
 /* --- Select a good internal scaling value for the FFT -----------*/
 /*                                                                  */
 /* The value of the parameter minscale is related to the amplitude */
 /* of the input signal.  This function assumes that the input      */
 /* from the pick-up is DC filtered.  It looks at the start of the  */
 /* acquisition and records the largest value.  It also looks at    */
 /* the end in case of beam blow-up.                                 */
 /*-----------------------------------------------------------------*/

 int    bestamp = -10000, myscale, *xstart, *xstop;

 /* Find largest +ve value, either at front or end of dataset */

 for (xstart = &x[0], xstop = &x[0] + (int)(size) - 10;
 xstart < &x[0xa]; xstart++, xstop++) {
     if (*xstart > bestamp) bestamp = *xstart;
     if (*xstop > bestamp) bestamp = *xstop;
 }

 /* Select minscale - depends upon FFT size also. */

 switch (size) {

     case    256:    if (bestamp < 220)
                             myscale = 0;
                        else if (bestamp < 420)
                             myscale = 1;
                        else if (bestamp < 880)
                             myscale = 2;
                        else
                             myscale = 3;
                        break;

     case    512:    if (bestamp < 110)
                             myscale = 0;
                        else if (bestamp < 210)
                             myscale = 1;
                        else if (bestamp < 420)
                             myscale = 2;
                        else if (bestamp < 880)
                             myscale = 3;
                        else if (bestamp < 1800)
                             myscale = 4;
                        else
                             myscale = 5;
                        break;

     case    1024:   if (bestamp < 110)
                             myscale = 1;
                        else if (bestamp < 210)
                             myscale = 2;
                        else if (bestamp < 420)
                             myscale = 3;
                        else if (bestamp < 880)
                             myscale = 4;
                        else if (bestamp < 1800)
                             myscale = 5;
                        else
                             myscale = 6;
                        break;
```

```
    default :       break;
 }
 *(acqdata) = myscale;
 *(acqdata + 1) = bestamp;
 return *acqdata;
}


int fft(size)
int size;
{
 /* --- Execute Fast Fourier Transform ----------------------------*/
 /*                                                               */
 /* The FFT functions are written in assembler and are contained  */
 /* within my library dspfft.lib                                  */
 /*---------------------------------------------------------------*/

 int    acqdata[2];

 bestmscl(size, acqdata);     /* Get miscale value                */
 zclear();                    /* Clear scaling pointers (hardware) */
 sect2 = stvmd(SECT_2 + ILI_ON + FAST);         /* Fastest DSP mode */

 switch (size) {

    case    256:    /* 256 point magnitude squared : 0.437ms */

                       fftm256(acqdata[0]);
                       break;

    case    512:    /* 512 point magnitude squared : 0.628ms */

                       fftm512(acqdata[0]);
                       break;

    case    1024:   /* 1024 point magnitude squared : 1.110ms */

                       fftm1024(acqdata[0]);
                       break;

    default :       /* Error : Illegal FFT size in command list */

                       acqdata[0] = 0x1111;
                       break;
 }

 return acqdata[0]; /* return minscale (used for re-scaling) */
}


float qmeas(size)
int size;
{
 /* --- Calculate the q value from a magnitude(2) spectrum ---------*/
 /*                                                                */
 /* Calculates the q of the PS.  Interpolation method is applied to */
 /* the spectrum of a 512-point magnitude-squared FFT.  If the size */
 /* of the main peak is < QTHRESH no q-calculation is performed. In */
 /* this case, it returns q = 0.0.                                 */
 /*----------------------------------------------------------------*/
```

```
int    acqdata[2], p1, situ;
double a1, a2, a3, points = (double)(size);
float  qout = 0.0;      /*  Indicates low spectral amplitude if
                            returned */

zclear();
bestmscl(size, acqdata);

if (acqdata[1] > QTHRESH) {

    sect2 = stvmd(SECT_2 + ILI_ON + FAST);  /* Fast DSP mode */
    fftm512(acqdata[0]);
    p1 = 1 + peaksrch(y + 1, 255);  /* ignore DC (upto bin 20) */

    /* If the peak is near dc, I do not calculate Q */

    if (p1 < 20) {
        /* Do nothing */
    }
    else {

        /* Look at situation before interpolating */

     if (y[p1 + 1] > y[p1 - 1])       /* True peak to right of p1. */
         situ = 1;
     else if (y[p1 - 1] > y[p1 + 1])  /* True peak to left of p1.  */
         situ = 2;
     else if (y[p1 + 1] == y[p1 - 1]) /* Low spectral resolution,  */
         situ = 3;                    /* hence p2 & p3 are equal.  */
        else if (y[p1] == y[p1 + 1]) /* Low res', but p1 & p2 are */
         situ = 4;                    /* now equal in magnitude.   */

     /* The peak search looks from the end of the vector towards
      * the start.  Hence if two peaks have the same magnitude,
      * the position of the one on the left is returned.
      */

     a1 = sqrt((double)(y[p1])); /* Take root, o/p mag squared */

     switch (situ) {

         case 1:  /* Interpolate for true peak right of p1 */

                 a2 = sqrt((double)(y[p1 + 1]));
                  a3 = sqrt((double)(y[p1 - 1]));
                  qout = (float)(((double)(p1) +
                  a2 / (a1 + a2)) / points);
                  break;

         case 2:  /* Interpolate for true peak to left of p1 */

                  a2 = sqrt((double)(y[p1 - 1]));
                  a3 = sqrt((double)(y[p1 + 1]));
                  qout = (float)(((double)(p1)
                         - a2 / (a1 + a2)) / points);
                  break;

         case 3:  /* Since p2 = p3, peak is almost exactly between
               * them. So, best bet is take p1 as the true peak.
               */

                  qout = (float)((double)(p1) / points);
                  break;
```

```
          case 4:  /* Since p1 = p2, best choice is half way */

                   qout = (float)(((double)(p1) + 0.5) / points);
                   break;

          default  :  /* Error : Unknown situation */
              qout = 0.0;
           break;
          }
      }
 }
 return qout;
}


main()
{
 int            *pcmd, number_cmds, countfft, ffttemp[1024], i;
 unsigned int   sfft_start, bank_end = 0x6c00;
 struct cmdlist localcmd[5];
 int            qstore[MAXMEAS], dummy;

 /* Set bank operation mode and current bank, etc. */

 setarb(2);
 setbnk(A);
 roi(0, MASTER);     /* enable tri-state MODE signal */

 /*--------------------*/
 /* MAIN PROGRAM LOOP */
 /*--------------------*/

 while (MYTRUE) {

     setbnk(A);
     hostf();     /* Wait for new list of instructions */

     /* Receive measurement list into data bank B */

     *pcmd = (int)(&globalcmd[0] - VMEOFFSET);
     flagt();
     dputd(pcmd, 1);                      /* Pass address to host    */
     hostf();                             /* Wait for data transfer  */
     swpbnk();                            /* Let Texas access bank B */
     memcpy(localcmd, globalcmd, 5);      /* Make a local copy       */
     number_cmds = number[0];

     /* If 1st measurement type is SLIDING FFT, do it now.  If not,
      * read through the struct localcmd[] & execute sequentially
      */

     if (localcmd[0].kind == MTSFFT) {

          /* Acquire 2 * 0x6800 bytes from 0x8400 upwards */

          sfft_start = 0x400;
          insmax(contint);
          mbase(sfft_start);
          setcern(CONT);
          getcern(FULLACQ);
          mwait(); /* End of whole acquisition here */
```

```
/* x[0] -> x[0x3FF] is FFT input window, so move data there */
/* But since the data is in two banks, we need to juggle it */
/* about sometimes.                                         */

     for (countfft = 0; countfft < localcmd[0].quantity;
     countfft++) {
          if (sfft_start <= bank_end - localcmd[0].fftsize) {
               memcpy(&x[0], &x[sfft_start], localcmd[0].fftsize);
          }
          else if (sfft_start > (bank_end - localcmd[0].fftsize)
               && sfft_start < bank_end) {
               memcpy(&x[0], &x[sfft_start], bank_end -
                      sfft_start);
               swpbnk();
               memcpy(&ffttemp[0], &x[0x400],
               localcmd[0].fftsize - (bank_end - sfft_start));
               swpbnk();
               memcpy(&x[bank_end - sfft_start], &ffttemp[0],
               localcmd[0].fftsize - (bank_end - sfft_start));
          }
          else if (sfft_start >= bank_end) {
               swpbnk();
               sfft_start = (sfft_start - bank_end) + 0x400;
               memcpy(&x[0], &x[sfft_start], localcmd[0].fftsize);
          }

     /* Having got the correct data in the window, run the FFT and
      * build the result header.
      */

          globalres[0].fftmscl  = fft(localcmd[0].fftsize);
          globalres[0].kind     = MTSFFT;
          globalres[0].ressize  = localcmd[0].fftsize / 2;
          globalres[0].respntr  = (int)(&y[0]);

          /* Send an interrupt to the DSC */

          swpbnk();
          user(0);    /* Interrupt source */
          *pcmd = (int)(&globalres[0] - VMEOFFSET);
          dputd(pcmd, 1);
          hostf();
          swpbnk();
          sfft_start += localcmd[0].slidstep;
     }
}

/* Perform Sliding FFT acquisition only, and give */
/* raw acquisition data... not the spectra        */

else if (localcmd[0].kind == MTRAWSFFT) {

     insmax(fullint);
     mbase(0);
     setcern(CONT);
     getcern(ALLBANK);
     mwait(); /* End of whole acquisition here */

     /* Send an interrupt to the DSC */

     globalres[0].fftmscl  = 0;
     globalres[0].kind     = MTRAWSFFT;
     globalres[0].ressize  = 0xffff;
```

```
          globalres[0].respntr  = (int)(&x[0] - VMEOFFSET);

          swpbnk();
          user(0);    /* Interrupt source */
          *pcmd = (int)(&globalres[0] - VMEOFFSET);
          dputd(pcmd, 1);
          hostf();
          swpbnk();
          flagt();
          hostf();
     }

     /* Execute measurements other than the Sliding FFT */

     else {
          /* Acquire enough data for the first measurement */

          insmax(aint);
          myflag = 0;
          setcern(SING);
          mbase(x - 0x8000);
          getcern(~(localcmd[0].fftsize) + 1);
          while (myflag == 0) {}

/* Concurrent processing/acquisition for remaining measurements */

          for (i = 0; i < number_cmds; i++) {

               /* Acquisitions for subsequent measurements */

               swpbnk();
               if (i < number_cmds - 1) {
                    insmax(bint);
                    mbase(x - 0x8000);
                    myflag = 1;
                    getcern(~(localcmd[0].fftsize) + 1);
               }

               /* During this acquisition, process previous data */

               switch (localcmd[0].kind) {

                    case    MTFFT:  /* Real block floating point FFT */

                         globalres[0].fftmscl =
                              fft(localcmd[0].fftsize);
                         globalres[0].kind = MTFFT;
                         globalres[0].ressize = (localcmd[0].fftsize
                              /2);
                         globalres[0].respntr = (int)(&y[0] -
                              VMEOFFSET);
                         while (myflag == 1) {}

                         /* Send interrupt */

                         user(9);
                         swpbnk();
                         *pcmd = (int)(&globalres[0] - VMEOFFSET);
                         dputd(pcmd, 1);
                         hostf();
                         swpbnk();
                         break;
```

```
                case    MTRAWFFT:  /* Real FFT */

                     globalres[0].fftmscl = 0;
                     globalres[0].kind = MTRAWFFT;
                     globalres[0].ressize = (localcmd[0].fftsize);
                     globalres[0].respntr = (int)(&x[0] -
                          VMEOFFSET);
                     while (myflag == 1) {}

                     /* Send interrupt */

                     user(9);
                     swpbnk();
                     *pcmd = (int)(&globalres[0] - VMEOFFSET);
                     dputd(pcmd, 1);
                     hostf();
                     swpbnk();
                     break;

                case    MTQ:       /* q measurement */

                     /* store value in buffer */

                     qstore[i] = (int)(65536.0 *
                          qmeas(localcmd[0].fftsize));
                     break;

                default   :  /* Error : Unknown measurement type */

                     globalres[0].respntr = 0x0;
                     globalres[0].kind = 0x2222;
                     globalres[0].ressize = 0x0;
                     globalres[0].fftmscl = 0x0;
                     while (myflag == 1) {}
                     swpbnk();
                     *pcmd = (int)(&globalres[0] - VMEOFFSET);
                     user(10);
                     dputd(pcmd, 1);
                     hostf();

                     swpbnk();
                     break;
           }
      }
}

/* Transfer q values */

if (localcmd[0].kind == MTQ) {

     /* Copy array of q values into global memory */

     for (dummy = 0; dummy < number_cmds; dummy++) {
          x[dummy] = qstore[dummy];
     }

     globalres[0].kind = MTQ;
     globalres[0].ressize = number_cmds;
     globalres[0].fftmscl = 0;
     globalres[0].respntr = (int)(&x[0] - VMEOFFSET);
     user(10);
     swpbnk();
```

```
          *pcmd = (int)(&globalres[0] - VMEOFFSET);
          dputd(pcmd, 1);
          hostf();
          swpbnk();
     }
 } /* FOREVER LOOP */

 v16_exit();
}


/*--------------------------------------------------------------------*/
/* psvasp.h                                                           */
/*                                                                    */
/* Header info for the VASP16 module psvasp.c                         */
/*--------------------------------------------------------------------*/
/* Created : September 1992                    S. Johnston, PS/BD.    */
/* Version : 1.0                                                      */
/*--------------------------------------------------------------------*/

/*--- My special constants -------------------------------------------*/

#define SING          0x8  /* Single trigger mode                     */
#define CONT          0x0  /* Double trigger (1 false) mode           */
#define QTHRESH         5  /* Amplitude limit for Q calculation       */
#define DATALOC     0x400  /* Location of Sliding FFT data in x[]     */
#define FULLACQ    0x9800  /* 2s compliment of 0x6800                 */
#define ALLBANK    0x8000  /* 2s compliment of 0x8000                 */
#define VMEOFFSET  0x8000  /* EG *pcmd = (int)(&x[0] - VMEOFFSET);    */
#define MYTRUE          1  /* Forever loop tests this value           */
#define MAXMEAS       300  /* Maximum number of meas allowed          */
#define MTFFT           0 /* Measurement id types                    */
#define MTSFFT          1  /*          "                  */
#define MTRAWFFT        2  /*          "                  */
#define MTRAWSFFT       3  /*          "                  */
#define MTQ             4  /*          "                              */

/*--- Global memory declarations -------------------------------------*/
/*                                                                    */
/*--- NAME ---------------------- SIZE ----- ADDRESS ---- PURPOSE --*/
asm("_x         .usect  .global, 6c00h"); /* 0x8000, FFT input data */
asm("_y         .usect  .global,  800h"); /* 0xec00, FFT output     */
asm("_t         .usect  .global,  800h"); /* 0xf400, FFT temp       */
asm("_sect2     .usect  .global,    1h"); /* 0xfc00, VSP mode word  */
asm("_max       .usect  .global,    8h"); /* 0xfc01, Zoran storage  */
asm("_scale     .usect  .global,    4h"); /* 0xfc09, Zoran storage  */
asm("_number    .usect  .global,    1h"); /* 0xfc0d, No. commands   */
asm("_globalcmd .usect  .global,    5h"); /* 0xfc0e, Command List   */
asm("_globalres .usect  .global,    5h"); /* 0xfc13, Result header  */
asm("_myflag    .usect  .global,    1h"); /* 0xfc18, Acq complete   */
asm("_debug     .usect  .global,   20h"); /* 0xfc19, debugging      */

extern int  x[], y[], t[], sect2, number[], max[], scale[], myflag,
debug[];

/*--- Structure for measurement instructions -------------------------*/

struct cmdlist {
     int     kind;       /* q measurement, fft or sliding fft */
     int  fftsize;       /* 256, 512 or 1024 point fft choice */
     int quantity;       /* number of ffts (sliding fft only) */
     int slidstep;       /* step-interval  (sliding fft only) */
} extern globalcmd[];
```

```
/*--- Structure for results header ---------------------------------*/

struct reslist {
     int     kind;          /* q measurement, fft or sliding fft */
     int  fftmscl;          /* scaling for fft output spectrum   */
     int  ressize;          /* size of result data               */
     int  respntr;          /* pointer to result data            */
} extern globalres[];

/*-------------------------- EOF ----------------------------------*/
```

A.3. DSPFFT LIBRARY CONTENTS

This library was created using DSPAR [(c) Texas Instruments] and contains the FFT functions, CERNbus routines and a user interrupt function.

```
DSP Archiver                    Version 5.20
(c) Copyright 1987, 1989, Texas Instruments Incorporated

         FILE NAME    SIZE   DATE
  ----------------    -----  ------------------------
       fftm256.obj     2111  Fri Feb 28 09:18:34 1992
       fftm512.obj     2481  Fri Feb 28 09:18:24 1992
      fftm1024.obj     3387  Fri Feb 28 09:18:12 1992
        fft256.obj     2103  Fri Feb 28 09:17:36 1992
        fft512.obj     2465  Fri Feb 28 09:17:26 1992
       fft1024.obj     3315  Fri Feb 28 09:17:16 1992
       setcern.obj      288  Wed Apr 08 09:42:34 1992
       getcern.obj      336  Wed Apr 08 09:44:24 1992
       swaparm.obj      328  Sat Aug 04 16:28:22 1990
       singacq.obj      324  Fri Apr 03 14:49:12 1992
          user.obj      322  Tue Apr 27 10:32:52 1993
```