Daresbury Laboratory
Technical Manual

EC738 , 32 CHANNEL SCALER

Instrumentation Laboratory
ROOM B22
SERC
Daresbury Laboratory
Daresbury,
Warrington
England
WA4 4AD
Tel 0925 603250

Revision 5.0

Contents

Overview 3
Specification 4
Scaler Inputs 4
Control Inputs 4
Internal switches 4
Base address, Interrupt vector 4
Interrupt level 4
Control Input Levels 4
Scaler Input Levels 4
VME Interface 4
Registers 5
Memory 5
Interrupts 6
General 6
Programming Information 7
Memory layout 7
Scalers 8
Initialize Command 9
XFER Command. 9
Clear IRQ Command 9
Control/Status Register 10
Interrupts 11
Principles of Operation 13
Description 13
Block Diagram. 13
Circuit Description 14
Scalers 14
Memory 14
Adder 15
Address Generator 15
Control. 15
VME Interface 15
Connectors 16
Time frame input, 14 WAY 3M 16
Scaler Inputs. 17
Configuration of the Scaler Inputs 18
TTL Compatible Inputs. 18
ECL Compatible Inputs. 18
Device Driver Library. 19
Example program 24

Overview

This module provides 32 high speed counting channels, each with a capacity of 24 bits which allows over 16 million counts. Each scaler is connected to a 1024 time frame by 24 bit memory. The contents of each scaler can be added to its memory allowing the data to be accumulated into time frames over many cycles.

Inputs are normally supplied configured for TTL signals but there is an option for ECL inputs. Maximum counting rates are governed by the choice of driver and receiver device. With TTL 74F devices, clock rates up to 100 MHz are possible. With ECL this can be extended to 150 MHz.

Gating of counting is simultaneous on all 32 channels. It can be controlled via software or a front panel VETO input.

The transfer of the scalers to the time frame memory is triggered by software or a front panel XFER input. This causes the module to inhibit counting for 500 nS, while the scaler contents are transferred to the scaler shadow registers. The scalers then are cleared and counting can resume. It takes an additional 16 µS to add the data from the 32 shadow registers to the memory locations addressed by a front panel time frame input.

When combined with an EC740 Time Frame Generator, the two modules can collect data for a time resolved experimental run with up to 1024 time frames repeated for up to 4096 cycles, without any external or CPU intervention.

Inputs are available on the TFG to trigger the modules from external sources. The TFG provides eight programmable pulse outputs to trigger the experiment.

Interrupts are available from the scaler to warn if the scalers or memory reach half full. Other interrupts are available from the TFG, showing the end of cycle, end of run etc. - see TFG manual.

The module is implemented on a 6U VME card as a VME bus slave. It is controlled by and the data is read out using the VME bus.

Specification

Scaler Inputs

Inputs 32 inputs, Differential ECL or single ended TTL on 2 off, 34 way 3M type connector.
Scaler increments on the positive going edge.

Impedance 100 ohm to -2v with Differential ECL, 100 ohm to +3v with TTL.

Pulse width 5 nS minimum (using 74F inputs).

Double pulse 10 nS. resolution

Maximum Rate 100 MHz using 74F inputs, 150 MHz using ECL.

Control Inputs

VETO Active Low Veto Signal => Do not count.

XFER Causes transfer of data from scalers to memory.

These signals are either TTL or Fast NIM levels, as set by switch E14.

Internal switches

Base address, Interrupt vector

This is set by two hex switches E18 and E19. If the switches are set to e.g. 0x3A, then the memory base address is 0x3A0000 and the control register base address is set to 0x3A00 (often accessed using 0xFFFF3A00). The interrupt vector will also be 0x3A. Note that the base addresses must be set to even numbers e.g. F0, F2, F4 ....

Interrupt level

The modules interrupt level is set using the DIL switch E46. Correct settings are achieved by setting one and only one switch to on.

e.g. IRQ's on Level 3 :-

IRQ6 off
IRQ5 off
IRQ4 off
IRQ3 on
IRQ2 off
IRQ1 off

Note that IRQ7 is not available.

Control Input Levels

The front panel inputs XFER and VETO can be either TTL or Fast NIM. This is controlled by E14

Scaler Input Levels

To enable ECL inputs to the scalers, several changes have to be made - see later. The termination resistor level is set by switch E13.

VME Interface

This module is a VME slave and interrupter. It occupies to two areas of the VME memory. The scalers and control registers are in A16 (short) address space and allow D32/D16 access. The memory is in A24 (standard) address space and also allows D32/D16 access. The base addresses for both regions are derived from the same 8 bit number, the

module id, selected by two hex switches E18 & E19.

Registers

The module uses 256 bytes of A16 (short) address space to address the scalers, status register and the initialise, transfer and clear IRQ commands. The base address for the registers is module_id*0x100.

The register offsets, sizes and accesses are show in the table below.

Offset	Data size	Read Action	Write Action
0x0	32/16	Read Scaler 0 24 bits (D23..D0) valid.	Send test clock to all scalers. Data ignored.
0x4	32/16	Read Scaler 1	Send test clock to all scalers. Data ignored.
0x8	32/16	Read Scaler 2	Send test clock to all scalers. Data ignored.
....	....	....	....
0x7C	32/16	Read Scaler 31	Send test clock to all scalers. Data ignored.
0x83	8	Read Status Register	Write Control/Status Register.
0x87	8	None	Transfer Scaler data to memory.
0x8B	8	None	Clear Interrupt Request
0x8F	8	None	Initialise module.

Memory

The module requires 128k byte of A24 (standard) address space to address the memory. The base address is given by module_id*0x10000. Note that this suggests that the base address can be set on 64k boundaries. However, since the memory is 128k bytes long, only even module ids should be used, giving 128k boundaries. D16 and D32 read and write accesses are allowed.

Bits	Values	Description
A23..17	0..0x7F	Module Id, should be 0,2..0xFE
A16..7	0..1023	Time frame
A6..2	0..31	Scaler channel
A1..0	0..3	Byte address Normally use address 0, with long word access.

The accumulated scaler data is arranged on 32 bit boundaries so that incrementing

addresses scan through scalers and then time frames.

Note. As scalers and Memory are 24 bits wide, 32 bit reads will give bits D24 - D31 as zero's.

VME read-out of the module may occur at any time. It is however given second priority to new data transfers. If the module is busy the VME cycle will be delayed by up to 16µs until the transfer has completed.

It accepts address modifiers, Data access NP and SU (29h,2Dh,39H and 3Dh).

Interrupts

Interrupts are generated on IRQ line 1 to 6 . There are two interrupt conditions, Scaler Half Full and Memory Half Full. The Interrupt Status/ID byte is set using the base address switch.

General

Size	Single width 6u VME Unit.
Power	+5 V @ 2400 mA. -12 V @ 1500 mA. (With Scaler inputs ECL). -12 V @ >10 mA. (With Scaler inputs TTL).

Programming Information

Memory layout

The memory consists of 24 bit words on D23..0 arranged on 32 bit boundaries. When read out the upper 8 bits return 0. Therefore the data can be read and written using 32 bit instructions. To read and write blocks of memory, using the device driver use:-

```
long *buffer
int path;              /* Path number returned by     */
                       /* open("/mcs0",0);            */
int offset;            /* Offset from start of memory */
                       /* in long words               */
int npts;              /* Number of long words to     */
                       /* read/write                  */
..............................
mcs_wrmem(path, offset, npts, buffer);
..............................................
mcs_rdmem(path, offset, npts, buffer);
```

Since data is accumulated into the memory, it should initially be cleared by writing zeros to it. Using the device driver, this is done using

```
mcs_clrmem(path);
```

When the module performs an XFER command, the contents of the scalers are added to the memory. The layout of the scalers within the memory is given below:-

Offset	Description
0	Scaler 0, Time Frame 0
4	Scaler 1, Time Frame 0
8	Scaler 2, Time Frame 0
0x0C	Scaler 3, Time Frame 0
..	....
0x7C	Scaler 31, Time Frame 0
0x80	Scaler 0, Time Frame 1
0x84	Scaler 1, Time Frame 1
..	....
0xFC	Scaler 31, Time Frame 1
0x100	Scaler 0 , Time Frame 2
..	....
0x17C	Scaler 31, Time Frame 2
..	....
0x1FF80	Scaler 0, Time frame1023
0x1FFFC	Scaler 31, Time Frame 1023

To enable more efficient accessing of the desired parts of the scaler memory, various functions are provided which take account of this layout. Reading all (or some) of the scalers for a single time frame can easily be done using `mcs_rdmem()`. To read several time frames (starting from the first) for a single scaler channel use `mcs_rdcol()`.

If several scaler boards are being used, then the `mcs_rdrows()` functions should be considered. This function allows the contents of several boards to be interleaved so that reading from the first location of the output buffer, an incrementing pointer scans channel 0..31 for the first board, then 0..31 for the second boards, up to channels 0..nscalers-1 for the last board all for time fame 0. Then the incrementing pointer will scan through all the channels time frame 1 etc. To use this function:-

```
int num_boards;          /* number of boards */
int paths[MAXBOARDS];    /* path for each board */
int *buffer;
int board;
int first_tf;            /* first time frame to read */
int  num_tf;             /* number of time frames to read*/
....
for (board=0; board<num_boards;board++)
    mcs_rdrows(path[board], first_tf, num_tf,
               num_boards*32, 32, buffer+32*board);
```

The argument `num_boards*32` tells the device driver the total number of channels in the output buffer. If this is not a multiple of 32, some channels are not used on the last board and the nscalers argument would be reduced from 32 on the last call to `mcs_rdrows()`.

The memory can be read out while data collection is occurring but data is transferred from the scalers to the memory only after the XFER operation. For experiments with short time frames and many cycles of the TFG, the memory can be read out to give a real time display as the data is accumulated. With a single cycle, it is only possible to read out the previous time frame for a real time display, as the current frame will contain zero until it ends. If the time frame is long enough, the scalers can be read directly to achieve a real time display for the current frame.

Scalers

The 32 scalers can be read directly from the A16 address space. This is done using the mcs_rdscal() function.

```
int scalers[32];
int first;               /* First scaler, to read out     */
int  num;                /* Number of scalers to read out */
int path;
......
mcs_rdscal(path, first, num, scalers);
```

The scalers can be read when they are counting. This will not interfere with the counting. However, if the scaler receives a clock pulse at the beginning of the read cycle, then the data may be corrupt. Data read from active scalers can give a useful real-time display to show that an experiment is working, but final data should only be read once counting has been stopped.

The scalers also have a self test mode. Writing to any of the scalers, will cause ALL the scalers to enter test mode and increment by the test pattern. This means that they will increment by 0x010101. Once the scalers are in test mode, they can be read out or transferred to memory to test the board. The status bit D0 is set to show test mode has been entered. Test mode is reset by performing an initialise command. Using the device driver:-

```
int path;                /* Path as returned by open() */
int num_clocks;          /* Number of times to send test
                            clocks                     */
.......
mcs_tstclk(path, num_clocks);
```

There is no mechanism to write data into the scalers. With other scalers, it is common to perform timing by preloading a counter and then letting it count up until it stops. This is not possible with the EC738. When timing is required (usually), it is provided by using the EC740 Time Frame Generator, which can time up to 1024 different length time frames and specify the time frame number to the EC738 - see EC740 manual.

Initialize Command

This is a dataless command. It causes the status register to reset which disables interrupts and stops counting. It also clears the scalers. This is the usual way to clear the scalers, except when XFER is being used. Using the device driver library:-

```
mcs_init(path);
```

XFER Command.

The XFER command is combined with the front panel XFER input. Pulsing the front panel input or writing (any data) to the XFER register will cause the module to transfer the scalers to the memory and then clear the scalers. This means that the scaler contents are added to the contents of the row of 32 memory locations addressed by the front panel Time Frame input. Using the device driver:-

```
mcs_xfer(path);
```

Clear IRQ Command

The module is a release on register access interrupter. To release interrupt request, it is necessary to write to the clear IRQ register. Using the device driver, this is done in the interrupt service routine. See section on interrupts.

Control/Status Register

The Status register can be examined at any time for module status. The layout and access to the bits is given in the table below.

Bit	Name in scaler.h	Access	Description
D7	SCALHF	Read	Set if any scaler reaches half full.
D6	MEMHF	Read	Set if any memory location reaches half full.
D5	FPVETO	Read	Set if FVETO is high => Counting can be enabled - see also D1.
D4	LEMOTYP	Read	Set if the front panel inputs are using NIM.
D3	INPTYP	Read	Set if the scaler inputs are ECL
D2	IRQENB	R/W	Set to enable the scaler interrupts
D1	VETO	R/W	Read shows whether scaler are VETOed. Set => Scalers will not count. Write sets software VETO bit.
D0	ENBTST	R	Set when the scalers are in test clock mode

In more detail:-

D7 Scaler Half Full Read only Status bit, can cause interrupt
This bit is set when any of the 32 scalers reach half full, 8 million counts. It will remain set if the counters wrap round through 0. The assertion of Scaler Half Full will usually be an error/exception condition. The suggestion is that on detecting Scaler half Full, or receiving this interrupt, the controlling program will stop counting and read out the counters before they wrap round (a further 8 million counts). This is why the half full condition is detected. This bit is cleared only by an Initialize command. The counters have a maximum capacity of 16 million so there is plenty of time to acknowledge the interrupt and either set the scaler VETO bit, or stop the entire system by stopping the TFG.

D6 Memory Half Full Read only Status bit, can cause interrupt
This bit is set when the addition of the scalers into the memory results in a location becoming at least half full. It can be used instead of scaler half full when experiments are being performed with short time frames, but with many cycles. (see TFG manual). Again this can be used to stop collection before the data wraps round and becomes invalid. This bit is cleared only by an initialise command.

D5 Front Panel Veto Read only status bit.
FVeto shows the level of the front panel Lemo signal called Veto. FVETO goes to make up the master VETO in status bit D1, which includes vetoes from other sources. If FVETO is 1 then counting can be enabled. If it is 0 then the front panel veto input is low and counting is stopped.

D4 Lemo Level Read only status bit
This bit allows software to determine whether the front panel Lemos VETO and XFER are set to TTL or NIM signals. If set they will trigger on Fast NIM levels and if reset they will trigger on TTL signals. This is controlled by E14

D3 Scaler Level Read only status bit
This bit allows software to determine whether the Scaler inputs are set to TTL or ECL signals. If set, they will trigger on ECL levels and if reset they will trigger on TTL signals. This is controlled by E13

D2 Interrupt Enable Read/Write control bit.
This bit enables an interrupt on the Scaler Half Full or Memory Half Full condition. It can be set or cleared by software and is reset by the initialize command. Reading shows if interrupts are enabled.

D1 Veto Read status, Write control bit.
Reading this bit shows the status of the **overall Veto** signal. If it is set, then counting is disabled. The overall Veto signal is the logical OR of the front panel Veto signal and the software Veto control bit. When either is true (note front panel signal is low for true) counting is disabled.
Writing this bit specifies the control word Veto bit. Setting this bit will disable counting.

D0 Test Mode Read only status bit.
This bit is set when the scalers are written to applying a test clock to them. Reading shows if test mode has been enabled. It is reset by the Initialise command.

Note that the value read from D1 is different to that written. This means that software **cannot user read modify write** operations such as `AND #$FD,Status(a0)` to this register. Instead, the software should keep a copy of the status register.

The status register can be read using the `mcs_rdstat(path)` function. The constants defined in scaler.h allow interpretation of the value returned.

Using the device driver, the control register cannot be written directly. Instead four functions are provided. These use a copy of the control/status register to preserve the control word Veto bit. Counting is enabled and disabled using the functions `mcs_enable(path)` and `mcs_disable(path)`. The interrupt enable bit is set and reset using the functions `mcs_irqenb(path, procid)` and `mcs_irqdis(path);`

Since the internal Veto signal is the OR of the control bit veto and the front panel Veto, counting can be controlled by either. However, in systems which use the EC738 in conjunction with a EC740 Time Frame Generator, it is normal to enable the scalers before an experimental run is started and to allow the front panel VETO (from the TFG) to control the exact timing. This is all but mandatory when more than one EC738 is being used, because a software enable/disable cannot be performed on all boards at the same time, whereas the front panel veto from TFG will be simultaneous for all boards.

Interrupts

There are two possible interrupt conditions. These are Scaler Half Full and Memory Half Full. They are caused when any scaler reaches half counts, or addition into the memory reaches half counts. The interrupt is a warning that the counter/memory is about to overflow. The intended mode of operation is that when either interrupt occurs, action will then be taken to stop data acquisition before the scalers or memory does actually overflow. This may either be stopping the scalers using `mcs_disable(path)` or using `tfg_init()`.

Interrupts are enabled by writing to status D2 set and disabled bit resetting status D2 low. When either Scaler Half Full or Memory Half Full become true, the interrupter asserts the VME IRQ(1-6) line as set by switch E46. EC738 does not have the hardware capability to drive IRQ7. The selected IRQ line stays active until the interrupt is serviced. The interrupt service routine clears the interrupt by clearing bit 2 of the control/status register and then writing to the clear IRQ register.

Using the device driver, an interrupt service routine is provided which clears the interrupt and then sends a signal to the specified process. Interrupts are enabled by calling the `mcs_irqenb(path, procid, signum)` function. The argument procid is either the process id of the process to be signalled or 0. If 0 is supplied, the device driver determines the process id of the current process and sends the signal to this process. The argument signum specifies the signal number to be sent.

The user state intercept handler (or mainline code) can then read the status register to determine the interrupt source and then take the necessary action.

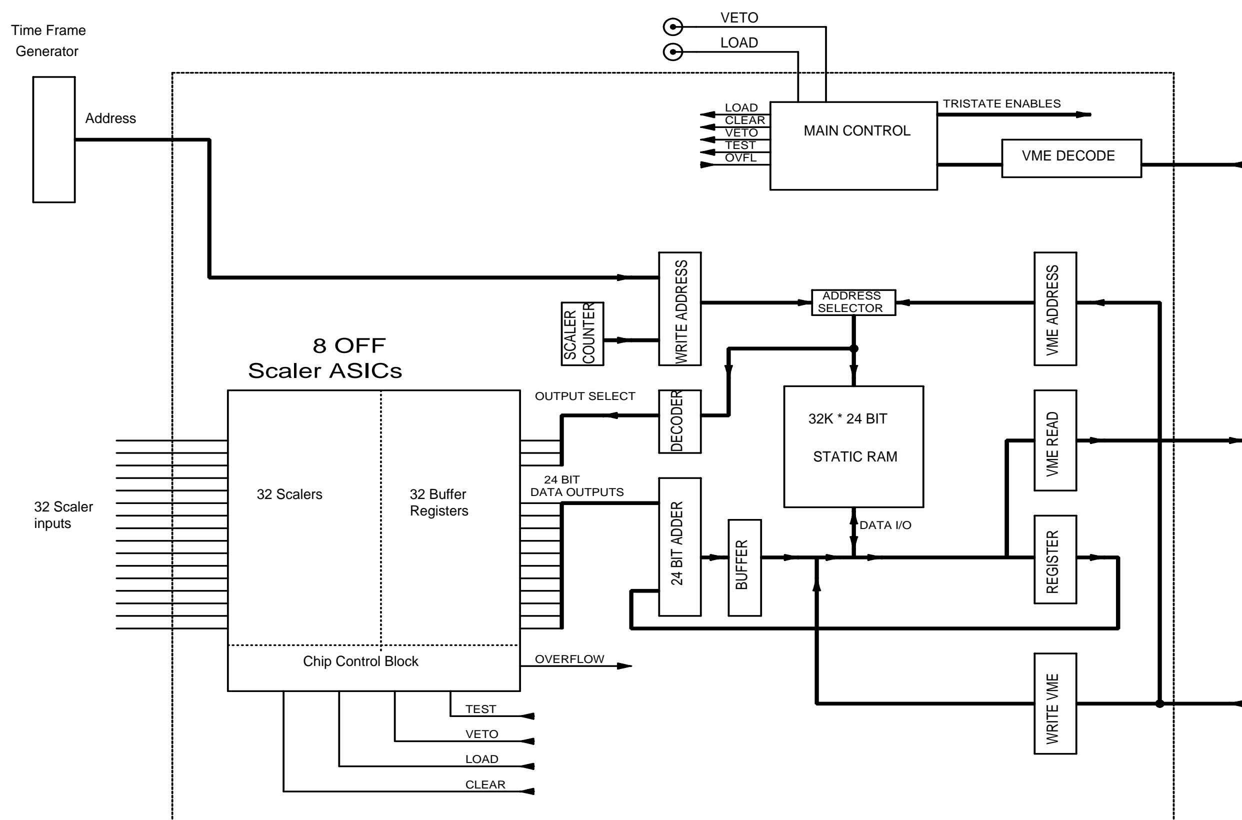
Principles of Operation

Description

This VME module is designed to count random pulses on 32 channels simultaneously, at rates of up to 150 MHz per channel. Each scaler has a capacity of 24 bits (over 16 million). A time framing input is provided which, when used in conjunction with the on board memory, can be used to frame the data into timed segments. The data for each frame is 32 words representing the integrated contents of the 32 scalers over the time period. During data collection the scalers or memory can be examined by reading them over the VME bus. In order to improved the statistical quality of the data collected over short time frames, the experiment can be repeated many times, with the on board memory providing an accumulation of all events for each particular time frame of the experiment. The time framing function is provided by the EC740 time frame generator module. A status flag is raised when either any scaler or any memory location is more than half full. This leaves plenty of time to terminate the experiment. The data can then be read-out via VME for display and storage.

Block Diagram.

The Block diagram shows the main features and data paths of the EC738.



Block Diagram of 32 Channel Scaler

Count pulses enter the module via two 34 way IDC input connectors. The input signals are terminated with Thevenin equivalent resistor packs and received by 74xx244 or 10125 receivers which drive the scalers. A clear signal is derived from either a VME initialise command or as part of a transfer cycle - see later. A Veto signal provides control over counting.

The Veto signal is derived from the front panel lemo signal and from a software controllable VETO bit. When either is true, counting is stopped. Note that when using TTL control inputs, the front panel Veto signal is active low.

Read-out of the scalers is always via their shadow registers. The shadow register takes a snap shot of the scaler when the load signal is used and holds the data during VME read cycles or transfer cycles.

During VME scaler read cycles, the scaler data is transfered to the shadow registers and is then added to zero before being driven on to the internal data bus and then out on to the VME bus. VME read cycles can be performed at any time and will not disturb counting. However, if a scaler is incremented just as its data is transferred to the shadow register, the data may be invalid. It is suggested that this small risk may be tolerated for real-time comfort displays, while the experiment proceeds. However, to ensure good data for storage it is necessary to veto counting.

When using the scalers with their on board memory, data is read-out of the scalers using a transfer command. This is caused by a front panel input or a VME command. When the transfer command occurs, the scalers are all vetoed for 500ns before transferring their contents to the shadow registers. This guarantees that any clock pulses have rippled through the counters , so ensuring that the data in the shadow registers is valid. A state machine then cycles through the 32 scalers. For each one, the data from the associated memory location is read into a register and then added to the contents of the shadow register and written back to the memory. The lower 5 address bits come from a counter. The upper 10 come from the time frame front panel input. This process takes 16 µs. However, since shadow registers hold the scaler contents while they are being stored in the on board memory, counting can resume after the data has been transferred to the shadow registers, thus minimising deadtime between frames.

Circuit Description

The Circuit Diagram can be found in the appendix. It consists of 6 basic sections. These are the Scalers, the Memory, the adder, the address generator, the Module Control and VME interface.

Scalers

The scalers are a block of eight ASIC's(E39, E40, E51, E52, E61, E62, E74, E75) . Each ASIC chip contains four scalers each with a shadow register. These can be controlled by the inputs: Clock, used as the counting input.; Veto, which gates the clock enabling or disabling counting; Load, which transfers the counter value into the associated shadow register; and Clear, which resets all counters to zero. There are four separate overflow outputs as well as an overall overflow. The 24 bit data from each scaler is selected by driving low one of four output selects together with the output enable.

Memory

The memory is a block of three static RAMs (E4,E5,E6), formed into a 32k by 24 bit memory. It is used as a local storage area, into which each scaler can store up to 1024 different readings before remote storage is necessary. During transfer cycles, the data is read into the adder's latch and then supplied from the adder output. For VME access, the internal data bus is connected to the VME data bus with D31..24 driven to zero (E60). There is provision to read D16..31 on D15..0, allowing D16 accesses (E9,E55).

Adder

The adder is used to add the scaler contents to the contents of the memory location being addressed and then store the result back into the same location. The circuit is built from six four bit adder's (E41, E42, E53, E63, E64, E76), a 24 bit input latch (E54, E65, E77) and a 24 bit output tristate driver (E43, E66, E78).

Address Generator

The Address Generator has two possible sources. During transfer cycles the address is provided by an internal counter and the time frame input. Durring VME cycles the memory is mapped directly into the VME A24 address space with the address being supplied from the VME address bus.

The VME address circuit consist of a sixteen bit latch with tristate buffers (E35, E36). The latch is loaded with the current VME address by the control circuit, when the module is addresses from the VME bus and it is not transferring data from the scalers to the memory.

The transfer address circuit has one eight bit counter (E3) with tristate buffer (E16) and a ten bit tristate register (E2). The counter is used to address the 32 scalers, so only 5 bit are used. This forms the lower 5 bits of an address word, with the time frame 10 bits forming the upper word.

Control.

The Control Circuit consists of one EPLD, an altera EP1800 (E31) and a PAL (E32). The EP1800 contains some VME decode and a state machine to control the operation of the scaler's memory and status register. The VME PAL part decodes all the VME cycles appropriate to the unit and supplies them as a coded request to the EP1800.

VME Interface

The VME interface into consists of Data bus buffers (E10, E11, E60, E67), control signal receivers (E23, E24), control signal drivers (E22) and address comparators (E21, E34). The interrupt vector is driven on to the data bus using E44.

Connectors

Time frame input, 14 WAY 3M

Pin	Signal	Pin	Signal
1	Time Frame Bit 0 (LSB)	2	Time Frame Bit 1
3	Time Frame Bit 2	4	Time Frame Bit 3
5	Time Frame Bit 4	6	Time Frame Bit 5
7	Time Frame Bit 6	8	Time Frame Bit 7
9	Time Frame Bit 8	10	Time Frame Bit 9 (MSB)
11	O/C	12	GND
13	GND	14	GND

Scaler Inputs.

34 way 3M IDC Connectors. Channel Numbers Bottom connector (Top connector)

Pin	Signal	Pin	Signal
1	+ve Channel 1(17)	2	-ve Channel 1(17)
3	+ve Channel 2(18)	4	-ve Channel 2(18)
5	+ve Channel 3(19)	6	-ve Channel 3(19)
7	+ve Channel 4(20)	8	-ve Channel 4(20)
9	+ve Channel 5(21)	10	-ve Channel 5(21)
11	+ve Channel 6(22)	12	-ve Channel 6(22)
13	+ve Channel 7(23)	14	-ve Channel 7(23)
15	+ve Channel 8(24)	16	-ve Channel 8(24)
17	+ve Channel 9(25)	18	-ve Channel 9(25)
19	+ve Channel 10(26)	20	-ve Channel 10(26)
21	+ve Channel 11(27)	22	-ve Channel 11(27)
23	+ve Channel 12(28)	24	-ve Channel 12(28)
25	+ve Channel 13(29)	26	-ve Channel 13(29)
27	+ve Channel 14(30)	28	-ve Channel 14(30)
29	+ve Channel 15(31)	30	-ve Channel 15(31)
31	+ve Channel 16(32)	32	-ve Channel 16(32)
33	GND	34	GND

Configuration of the Scaler Inputs

The scaler board is normally supplied configured for moderate speed TTL/CMOS inputs, which will allow for continuous count rates up to at least 25 MHz with the most generous noise margins. However, it is possible to change the inputs for either high speed TTL signals or (with care) for ECL signals. It is not possible to mix TTL and ECL inputs on one board (without modification).

TTL Compatible Inputs.

To use TTL levels for scaler inputs:-

Select TTL on switch E13.
Remove MC10125 at E26, E28, E48 ,E50, E58, E60, E71 & E73.
Insert 74xx244 at E37, E38, E68 & E69.
Insert patch headers at E26, E28, E48 ,E50, E58, E60, E71 & E73. Header has pins 2,6,10,14,& 16 connected together to ground.
Select resistor packs for positions E25, E27, E47, E49, E57, E59, E70 & E72. The standard resistor packs are 160/240 which give 96 Ω to +3V.

The input connectors are 34 Way IDC plugs. When the path headers are fitted as described above, every other strand of the IDC cable will be grounded. For short cable runs, it is possible to use standard IDC ribbon cable. However, for longer runs, in harsh EMC environments, it is also possible to use twisted pair ribbon cable or shielded ribbon cable. In many cases the 96Ω terminations will be satisfactory, though other 10 pin SIL termination packs can be fitted.

The choice of receiving device depend on the desired counting frequencies and minimum pulse widths. With 74HC244s the counting rates should be kept low. Pulse high and low times of at least 20 ns each are recommended. The use of HC inputs gives an input threshold of 2.5V. With such devices it is possible to driver the inputs from low powered devices such as 74HC[T] outputs.

If higher counting rates are desired (up to 100 MHz with short cables - derated for long cables), the use of 74F244s is recommended. The driving devices must then be able to sink at least 24 mA. High power devices such as the 74F244,74AS244,74AC244 etc. are recommended.

ECL Compatible Inputs.

To use differential ECL signals for scaler inputs

Select ECL on switch E13.
Remove 74244 from E37, E38, E68 & E69.
Insert MC10125 or MC10H125 at E26, E28, E48 ,E50, E58, E60, E71 & E73.
Select resistor packs for positions E25, E27, E47, E49, E57, E59, E70 & E72. The standard resistor packs are 160/240 which give 96 Ω to -2V. Note that this will appear as 192 Ω to the differential signal, which is normally too big. Alternative resistor packs are suggested with ECL.

In addition, consideration must be given to the power requirement for the -5.2 V rail. As standard, -5.2V is generated from the -12 V rail. This is sufficient to drive the receiver for the XFER and VETO inputs as TTL or fast NIM. However, when all 32 scaler channels and termination resistors are fitted for ECL operation, the regulator will require a heat sink fitting, increasing the module width to at least 2 units. Many VME crates cannot supply

much power from the -12 V rail and may not be able to supply more than one scaler board and possibly not even one. If desired, it should be possible to modify the board, to add a -5.2 V input. This would require modifications to the crate as VME does not specify a -5.2 V rail, so is not supplied as standard. Please consult Daresbury Laboratory before performing such a modification.

The exact value of the resistor pack will depend on the application and cable used. +5v, -5v and ground are available at the termination points. For error free counting it is important that the scaler signals are terminated correctly.

Device Driver Library.

The following describes how to communicate with the device driver, via OS/9 setstat from C. It is the only documented access point to the device driver.

```
**********************************************************************
*                            mcs_clrmem ()                           *
**********************************************************************

SYNOPSIS:      #include "scaler.h"

               int mcs_clrmem (path)
               int path;                              /* Path number */

FUNCTION:      Clear multi-channel scaler memory. Returns -1 on error
               & ERRNO for driver errors, else 0.


**********************************************************************
*                            mcs_disable ()                          *
**********************************************************************

SYNOPSIS:      #include "scaler.h"

               int mcs_disable (path)
               int path;                              /* Path number */

FUNCTION:      Disable the multi-channel scaler. Returns -1 on error &
               ERRNO for driver errors, else 0.


**********************************************************************
*                            mcs_enable ()                           *
**********************************************************************

SYNOPSIS:      #include "scaler.h"

               int mcs_enable (path)
               int path;                              /* Path number */

FUNCTION:      Enable the multi-channel scaler for data collection.
               Returns -1 on error & ERRNO for driver errors, else 0.


**********************************************************************
*                            mcs_init ()                             *
**********************************************************************

SYNOPSIS:      #include "scaler.h"

               int mcs_init (path)
               int path;                              /* Path number */

FUNCTION:      Initialise the multi-channel scaler. Returns -1 on error
               & ERRNO for driver errors, else 0.
```

```
*******************************************************************
*                        mcs_irqdis ()                            *
*******************************************************************

SYNOPSIS:       #include "scaler.h"

                int mcs_irqdis (path)
                int path;                           /* Path number */

FUNCTION:       Disable multi-channel scaler interrupts. Returns -1 on
                error & ERRNO for driver errors, else 0.


*******************************************************************
*                        mcs_irqenb ()                            *
*******************************************************************

SYNOPSIS:       #include "scaler.h"

                int mcs_irqenb (path, procid, signum)
                int path;                         /* Path number   */
                int procid;                       /* PID to signal */
                int signum;                        /* Signal to send */

FUNCTION:       Enable interrupts in multi-channel scaler. On interrupt
                a signal will be sent to the process id specified, which
                is the current process if 0. Returns -1 on error & ERRNO
                for driver errors, else 0.


*******************************************************************
*                        mcs_prcsig ()                            *
*******************************************************************

SYNOPSIS:       #include "scaler.h"

                int mcs_prcsig (path)
                int path;                           /* Path number */

FUNCTION:       Inquire which process the multi-channel scaler will
                signal on receipt of an interrupt. Returns -1 on error &
                ERRNO for driver errors, else process ID.


*******************************************************************
*                        mcs_rdcol ()                             *
*******************************************************************

SYNOPSIS:       #include "scaler.h"

                int mcs_rdcol (path, scalernum, npts, buff)
                int path;                           /* Path number */
                int scalernum;          /* Address of scaler <0 - 31> */
                int npts;      /* Number of points to read <1 - 1024> */
                int *buff;                /* Pointer to users buffer */

FUNCTION:       Read a column of multi-channel scaler memory into a
                buffer. Returns -1 on error & ERRNO for driver errors,
                else 0.
```

```
*********************************************************************
*                          mcs_rdmem ()                             *
*********************************************************************

SYNOPSIS:      #include "scaler.h"

               int mcs_rdmem (path, offset, npts, buff)
               int path;                              /* Path number */
               int offset;      /* Offset memory address <0 - 32767> */
               int npts;     /* Number of points to read <1 - 32768> */
               int *buff;                 /* Pointer to users buffer */

FUNCTION:      Read a block of multi-channel scaler memory into a buffer
               Returns : -1 on error & ERRNO for driver errors, else 0.


*********************************************************************
*                          mcs_rdrows ()                            *
*********************************************************************

SYNOPSIS:      #include "scaler.h"

               int mcs_rdrows (path, start_row, num_row, rowlen, nscaler,
buff)
               int path;                              /* Path number */
               int start_row;      /* Starting row number <0 - 1023> */
               int num_row;             /* Number of rows <1 - 1024> */
               int rowlen;/* Total length of spectrum <1 - 32*nscal> */
               int nscaler;    /* Number of scalers to read <1 - 32> */
               int *buff;                   /* Pointer to users data */

FUNCTION:      Read rows of memory, into a logically ordered block,
               corresponding to contiguous output from several modules.
               Such that output to disk is one write operation. Returns
               -1 on error & ERRNO for driver errors, else 0.

CAVEATS:       Note buffer layout

  scaler_1(0..31)  scaler_2(0..31) ... scaler_n(0..31) Time frame p
  scaler_1(0..31)  scaler_2(0..31) ... scaler_n(0..31) Time frame p+1
  scaler_1(0..31)  scaler_2(0..31) ... scaler_n(0..31) Time frame p+2
        .                .         ...            .
        .                .         ...            .
  scaler_1(0..31)  scaler_2(0..31) ... scaler_n(0..31) Time frame p+q-1

*********************************************************************
*                          mcs_rdscal ()                            *
*********************************************************************

SYNOPSIS:      #include "scaler.h"

               int mcs_rdscal (path, scalernum, npts, buff)
               int path;                               /*Path number */
               int scalernum;     /* Starting scaler number <0 - 31> */
               int npts;       /* Number of scalers to read <1 - 32> */
               int *buff;                 /* Pointer to users buffer */

FUNCTION:      Read the multi-channel scaler buffers to data array.
               Returns -1 on error & ERRNO for driver errors, else 0.
```

```
***********************************************************************
*                            mcs_rdstat ()                            *
***********************************************************************

SYNOPSIS:      #include "scaler.h"

               int mcs_rdstat (path)
               int path;                               /* Path number */

FUNCTION:      Read the multi-channel scaler status register
               Returns -1 on error & ERRNO for driver errors, else
               status register.


***********************************************************************
*                            mcs_tstclk ()                            *
***********************************************************************

SYNOPSIS:      #include "scaler.h"

               int mcs_tstclk (path, num_clk)
               int path;                               /* Path number */
               int num_clk;             /* Number of clock pulses >0 */

FUNCTION:      Send test clock pulse to multi-channel scaler.
               Returns -1 on error & ERRNO for driver errors, else 0.


***********************************************************************
*                             mcs_wrmem ()                            *
***********************************************************************

SYNOPSIS:      #include "scaler.h"

               int mcs_wrmem (path, offset, npts, buff)
               int path;                               /* Path number */
               int offset;   /* Offset address in memory <0 - 32767> */
               int npts;   /* Number of points to write  <1 - 32768> */
               int *buff;                /* Pointer to users buffer */

FUNCTION:      Write buffer to multi-channel scaler memory.
               Returns -1 on error & ERRNO for driver errors, else 0.



***********************************************************************
*                             mcs_wrrows ()                           *
***********************************************************************

SYNOPSIS:      #include "scaler.h"

               int mcs_wrrows (path, start_row, num_row, rowlen, nscaler,
buff)
               int path;                               /* Path number */
               int start_row;      /* Starting row number <0 - 1023> */
               int num_row;             /* Number of rows <1 - 1024> */
               int rowlen;/* Total length of spectrum <1 - 32*nscal> */
               int nscaler;   /* Number of scalers to write <1 - 32> */
```

```
                int *buff;                  /* Pointer to users data */

 FUNCTION:      Write rows of memory, from a logically ordered buffer,
                corresponding with contiguous input to several modules.
                Such that output to disk is one write operation.
                Returns -1 on error & ERRNO for driver errors, else 0.

 CAVEATS:       Note buffer layout

   scaler_1(0..31)  scaler_2(0..31) ... scaler_n(0..31) Time frame p
   scaler_1(0..31)  scaler_2(0..31) ... scaler_n(0..31) Time frame p+1
   scaler_1(0..31)  scaler_2(0..31) ... scaler_n(0..31) Time frame p+2
          .                .        ...             .
          .                .        ...             .
   scaler_1(0..31)  scaler_2(0..31) ... scaler_n(0..31) Time frame p+q-1


 ***********************************************************************
 *                           mcs_xfer ()                               *
 ***********************************************************************

SYNOPSIS:       #include "scaler.h"

                int mcs_xfer (path)
                int path;                             /* Path number */

 FUNCTION:      Transfer scaler buffers to multi-channel scaler memory.
                Returns -1 on error & ERRNO for driver errors, else 0.
```

Example program

```
/**************************************************************************
*
*   demo.c
*
*   This is a example program which uses the EC738 multichannel scaler
*   and the EC740 Time frame generator.
*   The user is asked to the describe the number and length of time frames.
*   Then the experiment is run.
*   The data written to an ASCII file scaler.dat
*
*   W. Helsby, Daresbury Lab 10/8/92
*
**************************************************************************/
#include <stdio.h>
#include <errno.h>
#include <tfg.h>
#include <scaler.h>


int tfg_signum=600;     /* Signal number sent by time frame generator   */
int tfg_signals = 0;    /* Number of signals received from tfg          */

/* Intercept routine detects signal from tfg and increments tfg_signals */
sig_handler(sig_code)
short sig_code;
{
    if (sig_code == tfg_signum)
        tfg_signals++;
    else if (sig_code == 2 || sig_code == 3)
        exit (sig_code);
}

main()
{
    int tfg_path, mcs_path;     /* path numbers for the two devices */
    FILE *ofp;                  /* FILE for output data             */
    register int frame, scaler;
    register unsigned long *lp;
    unsigned long *buffer;
    double ltime;
    int num_cycles, num_frames;

/* Allocate a buffer to store the data readout of the scalers */
    buffer = (unsigned long *)malloc(sizeof(unsigned long)*EC738_TOTSCA*
                                EC738_MAXFRAME);
    if (buffer == NULL)
    {
        fprintf(stderr,"Cannot malloc buffer for scaler data\n");
        exit(1);
    }
/* Open to these devices does not require read/write modes as all accesses
   are via setstat and getstat
*/
    if ((tfg_path=open("/tfg0",0)) < 0)
    {
```

```
        fprintf(stderr,"Cannot open path to /tfg0\n");
        exit (errno);
    }

    if ((mcs_path = open("/mcs0",0)) < 0)
    {
        fprintf(stderr,"Cannot open path to /mcs0\n");
        exit(errno);
    }

    if ((ofp=fopen("scaler.dat", "w")) == NULL)
    {
        fprintf(stderr,"Cannot open output file scaler.dat\n");
        exit (errno);
    }

    intercept (sig_handler);

    printf("Please enter live frame width in seconds :");
    scanf("%F", &ltime);

    printf("Please enter number of frame pairs :");
    scanf("%d", &num_frames);

    printf("Please enter number of cycles :");
    scanf("%d", &num_cycles);

/* Now setup the tfg */
    tfg_init(tfg_path);                       /* stops tfg if it is running,
                                                 clears status register     */
    setup_tfg(tfg_path, num_frames, ltime); /* See below */

/* Write the number of cycles required - 1 */
    tfg_wrcycle(tfg_path, num_cycles-1);

/* Enable interupts on the tfg at the end of the run only.
   Signal this process */
    tfg_irqenb(tfg_path, IRQEndRun, 0, tfg_signum);

/* Next setup the scalers */
    mcs_init(mcs_path);      /* Clears the scalers */
    mcs_clrmem(mcs_path);    /* Clears the memory  */
    mcs_enable(mcs_path);    /* Enables counting, once the tfg is started */

/*.................. printf("Mcs_status = 0x%X\n", mcs_rdstat(mcs_path));*/
.......................................................................
    printf("Starting experiment\n");
/* Now run the experiment */
    sigmask(1);
    tfg_start(tfg_path);

/* Wait for the experiment to finish and the tfg to give a signal */
/* During this period it would be possible to poll the tfg using :-
    tfg_rdstatus(tfg_path)
    tfg_rdframe(tfg_path)
    tfg_rdcycle(tfg_path);
    It is also possible to read scaler data from frames which have been
    completed using mcs_rdmem(...)  or to read the data currently being
    collected using mcs_rdscal(....)
```

```
*/

    sleep (0);
    if (tfg_signals == 1)
    {
        printf("Received tfg signal, experiment complete\n");
    }
    else
    {
        printf("Unexpected signal(s)\n");
    }

/* Now read all the scaler data into a buffer */
    mcs_rdmem(mcs_path, 0, EC738_TOTSCA*num_frames, buffer);
/* With multiple scaler boards the mcs_rdrows function may be more useful
   than mcs_rdmem, as it assembles the data into a contiguous block.
*/
    printf("Writing data to file scaler.dat ... ");
    fflush(stdout);
    fprintf(ofp,"   Frame   Scaler   Counts\n");
    lp = buffer;
    for (frame = 0; frame<num_frames; frame++)
    {
        for(scaler=0;scaler<EC738_TOTSCA;scaler++)
        {
            fprintf(ofp,"%8d %8d %8d\n", frame, scaler, *lp++);
        }
    }
    printf(" Done.\n");
    fclose(ofp);
}

/***************************************************************************
*
*   setup_tfg(path, num_frames, num_cycles, ltime)
*
*   This function fills the tfg memory with timing data to provide
*   num_frames of timing each of length ltime seconds.
*   The dead frames are all set to the minimum width of 10 us
*
***************************************************************************/
setup_tfg(path, num_frames, ltime)
int path, num_frames;
double ltime;
{
    unsigned short tfg_buff[EC740_MAXFRAME * 4];   /* 4 shorts per frame */
    register unsigned short *sp;
    register int frame;
    int lrate, lcount;      /* Live time expressed as a rate and a count */
    int drate, dcount;      /* Dead time expressed as a rate and a count */

/* Time frame widths are expressed as a 3 bits rate and 10 bit count field.
rate == 0 => 1 count ==  10 us
rate == 1 => 1 count == 100 us
rate == 2 => 1 count ==   1 ms
rate == 3 => 1 count ==  10 ms
rate == 4 => 1 count == 100 ms
rate == 5 => 1 count ==   1 s
rate == 6 => 1 count ==  10 s
```

```
rate == 7 => 1 count == 100 s

This routine is good for 10us < ltime < 20000s                         */
    lrate  = 0;
    lcount = (int)(ltime*100E3);
    while (lcount > EC740_MAXFCOUNT)
    {
        lrate++;
        lcount /= 10;
    }
    printf("Have chosen live rate = %d, count = %d\n", lrate, lcount);

/* Set dead frame to minimum width of 10 us */
    drate  = 0;
    dcount = 1;


/* Fill each frame pair with the dead and live frame timing and output
                                                                   data */
    sp = tfg_buff;
    for (frame = 0;frame <num_frames; frame++)
    {
        /* The lemo output data is just an incrementing count.
           Plug in LEDS to watch it */
        *sp ++ = drate << 10 | dcount;  /* Dead frame time as rate & count */
        *sp ++ = frame & 255;           /* Dead frame lemo output data     */
        *sp ++ = lrate << 10 | lcount;  /* Live frame time as rate & count */
        *sp ++ = frame & 255;           /* Live frame lemo output data     */
    }
    *(sp-1) |= MemEOF;                  /* Mark the last frame of the lap  */
    /* Note this mark (above the last frame lemo data word) is the way the
       number of frames information is stored */
    /* Write the buffer to the tfg.
       Note there are four short to each frame pair */
    tfg_wrmem(path, num_frames<<2, tfg_buff);
}
```