Daresbury Laboratory
Technical Manual

EC740 Time Frame Generator

Instrumentation Laboratory

ROOM B22
SERC
Daresbury Laboratory
Daresbury,
Warrington
England
WA4 4AD
Tel 0925 603250   Revision 2.0

Contents

Overview

This module generates a series of timing periods which are identified by a FRAME NUMBER. Each FRAME-PAIR is composed of a DEAD FRAME followed by a LIVE FRAME. The VETO output differentiates the live and dead part of the frame pair. Usually the module will be used in systems in which data is collected during the LIVE part of the frame while the DEAD frame allows time for sample recovery, stabilization etc. The module has up to 1024 time FRAME-PAIRS. The duration of each FRAME can be set from 10 μs to 102300 Seconds. Each FRAME can have a different time duration. The module can repeat the preset series of frames up to 4096 times. A series of FRAMES is referred to as a CYCLE (or LAP). A number of CYCLES would be an complete experimental RUN.

The module can be used to generate a series of frames of fixed timing to do a purely time resolved experiment. It is also possible to cause the TFG to pause between frames and then be re-triggered when the experiment is ready to continue. These pauses can be used to allow CPU intervention to adjust the experiment (such as in the calibration of the multi-wire linear system), or to allow the sample to recover to some condition (e.g. to a preset muscle tension) before continuing the exposure.

To enable control of sample conditions, 8 outputs are provided which can have a different level assigned to them in each time frame.

The module is setup from VME. This involves specifying the duration of each frame, the number of frames per cycle and the number of cycles per experimental run. In addition the output signal levels for each frame must be set if they are to be used. The status register must be setup to enable the front panel start and inhibit inputs if they are required. Similarly, any of the 4 possible interrupt sources must be enabled if required.

The module can then be started by either by a Front panel signal or a VME command. It can be made to pause after starting by a Front panel signal or by a VME command. After pause the module can be made to continue by the front panel start signal or a VME start command. The module can be reset at any time by issuing the VME Initialise command.

The module can generate an interrupt on Pause, End of LAP, End of RUN and Front panel Inhibit.

Specifications

Front Panel Inputs

Start TTL signal, edge active. Active edge selectable. Active Edge starts TFG, or continues if paused.

Inhibit TTL signal, edge active. Active edge selectable. Active Edge causes a pause request. At the beginning of the NEXT DEAD frame the TFG pauses.

Front Panel Outputs

Inhibit TTL signal, active level selectable. Asserted when the TFG is stopped (the reset state) and during dead frames. Negated during live frames

Framing TTL signal, active level selectable. Identical to Inhibit, but polarity is set independently.

Frame zero TTL signal, active level selectable. Negated when stopped (reset state). Asserted only during frame zero dead and live period. Negated during rest of run.

Address TTL, signal active high. 10 bit address which specifies the current frame number.

Outputs TTL, signal active level selectable. 8 off. These outputs are negated during the stopped (reset state). When running, they are loaded from the memory associated with each live and dead frame so can be different in every frame.

Leds

Inhibit Red, On while inhibited. (Reset and dead frames)
Framing Green, Flash for each new frame.
Frame Zero Yellow, On while in Frame zero.

Internal Switches

Dedicated Lemos, E23

E23, Selects Front Panel input active edges and Inhibit, frame 0 and framing output polarity.

Switch	Signal	OFF	ON
1	EXTSTART	Falling edge triggered	Rising edge triggered
2	EXTINHIBIT	Falling edge triggered	Rising edge triggered
3	FRAMING	High when live	High when reset or dead
4	FRAME 0	High during Frame 0	Low During frame 0
5	INHIBIT	High when live	High when reset or dead
6..8	UNUSED		

General purpose outputs, E29

E29, Selects polarity of general purpose Lemo outputs.

Switch	O/P	OFF	ON
1	PLS1	Low at reset or data=0	High at reset or data=0
2	PLS2	Low at reset or data=0	High at reset or data=0
3	PLS3	Low at reset or data=0	High at reset or data=0
4	PLS4	Low at reset or data=0	High at reset or data=0
5	PLS5	Low at reset or data=0	High at reset or data=0
6	PLS6	Low at reset or data=0	High at reset or data=0
7	PLS7	Low at reset or data=0	High at reset or data=0
8	PLS8	Low at reset or data=0	High at reset or data=0

VME Base Address

E19,E20 Selects Module base address in VME A24 address space.
These 2 hex DIL switches select the top 8 bits of the address A23..A16.
E.G. If E19,20 read E6 then the base address is 00E60000 hex.

VME Interface

This module responds to A24 D16 accesses only. The Address Map is as follows:-

Address bits	Valid Values	Description
A23..16	0..255	Module base adddress. Set by hex switches.
A15..14	00	Always zero
A13	0 1	Select Memory Select Registers
A12..2	0..2047	Select time frame
A1..0	0,2	Byte address

Byte Offset (HEX)	Description	Access
000000 - 0x1FFF	Memory Timing/port data (2K*2*16BitData)	Read/Write
0x2002	Current Frame Number.	Read only
0x2006	Cycle Number.	Write Number of cycles / read current cycle.
0x200a	Status/control	Read/Write
0x200e	Interrupt Vector and level.	Read/Write
0x2012	Start Module.	Data ignored.
0x2016	Pause Request.	Data ignored.
0x201e	Initialise.	Data ignored.

All access to the module is made in standard address space using 16 bit data. The software must not use 8 or 32 bit data operations.

General

The module requires 64k byte of standard addresses (A24 D16). It accepts address modifiers Data access NP and SU (39h & 3Dh).

Size   Single width 6u VME Unit.
Power   +5 V @ 800 mA.

Programming Information.

Memory layout.

The duration of each frame is stored in memory using two fields: count rate and number of counts. The count rate is given by a 3 bit field giving 7 possible count rates. 0 to 7 represents clock rate of 10us, 100us..100s). This is the rate at which the timer is decremented. The second field is a 10 bit number of counts which is loaded into the frame timer at the beginning of each frame. Note that 0 counts is not allowed. The two fields are stacked with the count at D0..9 and the rate at D10..12 to give a 13 bit word.

The memory is arranged into 1024 FRAME PAIRS. Each FRAME PAIR is describe completely by four 16 bit locations. Byte location 0 has the Dead Frame width, location 2 has the dead frame output port data plus pause and end of lap bits. Byte location 4 stores the width and location 6 port data for the LIVE frame.

Byte locations 0 (dead) and 4 (live), frame widths.

Data bits	Valid Values	Description
D12..10	0..7 0 1 2 3 4 5 6 7	Time unit length 0.01 ms 0.1 ms 1 ms 10 ms 100 ms 1 s 10 s 100 s
D9..0	1..1023	Number of time units to count

Byte locations 2 (dead) and 6 (live). Port, Pause and End of Cycle information.

Data bits	Valid Values	Description
D9	0 1	Continue End of Cycle
D8	0 1	Continue Pause at begining of frame.
D7..0	0..255	Port Data

Setting the pause bit in the data for a specific frame will cause the TFG to pause at the beginning of that frame. This means that the VETO line, frame address and port data are set for the frame the pause bit is in. The TFG then waits for an external or VME start signal. When the start signal is received, the TFG then waits for the time specified for the frame with the pause in it and then continues with the next frame. If a pause bit is put in the first dead frame, then the TFG will immediately pause when started. A second start command will make it continue by timing the first dead frame.

The end of cycle bit must be set in the last frame of the cycle. This is usually a live frame, but as far as the TFG is concerned it can be a live or dead frame.

Using the device driver, the usual way to setup the memory would be to assemble all the data in an array of unsigned shorts, and then use the `tfg_wrmem()` function. The data can be read back using `tfg_rdmem()`, if it is necessary to check the contents - usually just for testing.

```
unsigned short data[EC740_MAXFRAME*4];
int path;
int num_fpairs;
........
tfg_wrmem(path, num_fpairs*4, data);
........
tfg_rdmem(path, num_fpairs*4, data);
```

Frame Register.

The frame counter give has read only access. It is only meaningful to read the frame while the TFG is running or paused. To be consistent with the address offset into the memory data, the frame register is laid out as below.

Data bits	Valid Values	Description
D11..2	0..1023	Frame Number
D1	0 1	Dead frame Live frame
D0	0	Always Zero

Note however that the TFG must pre-fetch timing and port data before the end of the frame and hence the frame address counter is always one frame ahead of the current frame.

In this table there are n frame pairs, running from 0 .. n-1

Current Frame	Counter value	Description
First frame, (dead frame 0)	0x002	frame 0 live
Frame 0 live	0x004	Frame 1 dead
Frame 1 dead	0x006	Frame 1 live
Frame 1 live	0x008	Frame 2 dead
Frame 2 dead	0x00a	Frame 2 live
....	....	....
Frame n-1 dead	((n-1)<<2)+2	Frame n-1 live
Frame n-1 live	0	Frame 0 dead.

After the end of the last frame of the last cycle the counter shows frame n dead, with the current design, but this should be regarded as undefined.

Using the device driver this is accessed using `tfg_rdframe`.

```
tfg_frame = tfg_rdframe(path);
```

Cycle Register.

This register provided both read and write access, but it does not read back what was written.

The cycle counter should be written only when the TFG is stopped. The value written should be the number of cycles required - 1. Therefore 0..4095 => 1..4096 cycles. The value written is stored and is used every time the TFG is started.

The cycle counter should be read only while the TFG is running (or paused). The value read gives the current cycle but counts down from num_cycle-1 (during the first cycle) to 0 during the last cycle.

The table below defines the format of the data.

Data bits	Valid Values	Description
D11..0	0..4095	1..4096 cycles

Using the device driver:-

```
tfg_wrcycle(path, num_cycles-1);
current_cycle = tfg_rdcycle(path);
```

Control/Status Register.

This can be written to enable the external inputs and the required interrupts. It can be read to determine the module status and which interrupt source caused any interrupt.

BIT	Function.		Write/Read
D11	IRQ Status End of RUN	0 = Reset, 1 = Set	R
D10	IRQ Status End of LAP	0 = Reset, 1 = Set	R
D9	IRQ Status PAUSE	0 = Reset, 1 = Set	R
D8	IRQ Status Ext Inhibit	0 = Reset, 1 = Set	R
D7..6	Module State	00 = IDLE, 01 = Running 11 = Pause	R
D5	IRQ Enable RUN	0 = Disable, 1 = Enable	W/R
D4	IRQ Enable LAP	0 = Disable, 1 = Enable	W/R
D3	IRQ Enable PAUSE	0 = Disable, 1 = Enable	W/R
D2	IRQ Enable Ext Inhibit	0 = Disable, 1 = Enable	W/R
D1	Ext Inhibit Enable	0 = Disable, 1 = Enable	W/R
D0	Ext Start Enable	0 = Disable, 1 = Enable	W/R

The external inputs should only be enabled when the are required (to remove the risk of unexpected triggers).

D7 and D6 should be considered together. The possible states are IDLE, Running (not paused), Running but Paused.

NOTE that reading the status register will clear the interrupt status bits (D11..8)

Using the device driver the status register is setup using several commands. The external start and inhibit inputs are enabled using:-

```
tfg_enable(path, FPanelEnb);  /* Ext Start enabled */
tfg_enable(path, ExtInhEnb);  /* Ext Inh   enabled */
tfg_enable(path, FPanelEnb | ExtInhEnb);
                              /* Both enabled*/
```

To disable the front panel inputs use:-

```
tfg_disble(path);              /* Disable external inputs */
tfg_init(path);                /* Complete tfg initialise */
```

The interrupt enable bits cannot be set without specifying the process to be signalled. This is done using the `tfg_irqenb()` function -see section on interrupts.

Start Command

This is a dataless command. The data is ignored. Writing any short word to this address causes the TFG to start if it is stopped, or continue if it is paused.

Using the device driver:-

```
tfg_start(path);
```

Pause Request Command.

This is a dataless command. The data is ignored. Writing any short word to this address causes a TFG pause request. This is only sensible if the TFG is running. The pause request will cause the TFG to pause at the beginning of the NEXT dead frame.

Using the device driver:

```
tfg_pause(path);
```

Initialise Command.

This is a dataless command. The data is ignored. Writing any short word to this address causes the TFG to initialise. This stops the TFG, clears any interrupt requests and the status register. The front panel outputs are reset to VETO = TRUE and port values= 0, then >100 ns later, frame address = 0;

Using the device driver:

```
tfg_init(path);
```

Interrupts

Interrupts are generated on IRQ line 1 to 7. It is software selectable by writing to the INTRID register. Data bits 0-7 form the 8 bit Interrupt Vector with data bits 8-10 specifying the Interrupt level. It must be remembered however that interrupt level zero is not acceptable.

Data bits	Valid Values	Description
D10..8	1..7	Interrupt Level
D7..0	64..255	Interrupt Vector

The module is a ROAK, Release IRQ On interrupt AcKnowledge.

Using the device driver, it is not possible to change the interrupt vector/level. This information is loaded from the device descriptor when the device driver is INIZed and should then not be touched. When interrupts are enabled the device driver will send a specified signal to a specified process.

Interrupt Conditions

- **External Inhibit.** Interrupt caused on active edge or the front panel input, if the external inhibit input and the external inhibit interrupt are enabled.
- **Paused.** When enabled, this interrupt is generated whenever the module enters the paused state as a result of external inhibit, VME pause request or memory pause bit.
- **End of cycle.** When enabled, this interrupt is generated every time the TFG completes a cycle.
- **End of RUN.** When enabled this interrupt is generated when the TFG reaches the end of the last frame of the last cycle, i.e. the end of the run.

Interrupts are enabled using bits (D2..5) in the status register. When an interrupt occurs its associated status bit (D8..11) is set in the status register. Reading the status register enables the interrupt source to be determined. Note that this then clears D8..11 until another interrupt arrives.

The interrupts are edge active. This means that enabling e.g. enabling a pause interrupt when the module is paused will not cause an interrupt. The interrupts must be enabled before they occur.

If two interrupts sources (say end of cycle and end of run) occur together then there will be only one interrupt request cycle, but both bits will be set in the status register. If several interrupts occur in succession - such as several end of cycle interrupts - then the second potential interrupt will generate an interrupt cycle provided the previous interrupt acknowledge cycle has occurred. There is no need for any other CPU intervention (such as reading the status) to enable the second interrupt.

Using the device driver, interrupts are enabled using the `tfg_irqenb()` function. This requires four arguments. These are the path, enable bits for the four interrupt sources, the process id to be signalled and the signal to be sent on interrupt. If process 0 is specified, then the device driver signals the calling process. This is the usual case.

```
tfg_irqenb(path, IRQExtInh | IRQPause | IRQEndCycle |
IRQEndRun, process, signum);
```

Interrupts are disabled by a `tfg_init(path)` or by using the `tfg_irqdis(path)` command.

Principles of Operation

Description

Block diagram 1 shows the main features and data paths of the EC740. The Module is controlled from VME, although some commands are also enabled by front panel signals. The module has two MAX chips which control all the module functions such as VME data transfer and period timing . Max1 contains a VME DTB slave and interrupter and manages the internal buses and the memory. Max2 contains a divider chain plus counter to generate the time frames from information provided by Max1.

BLOCK DIAGRAM 1.
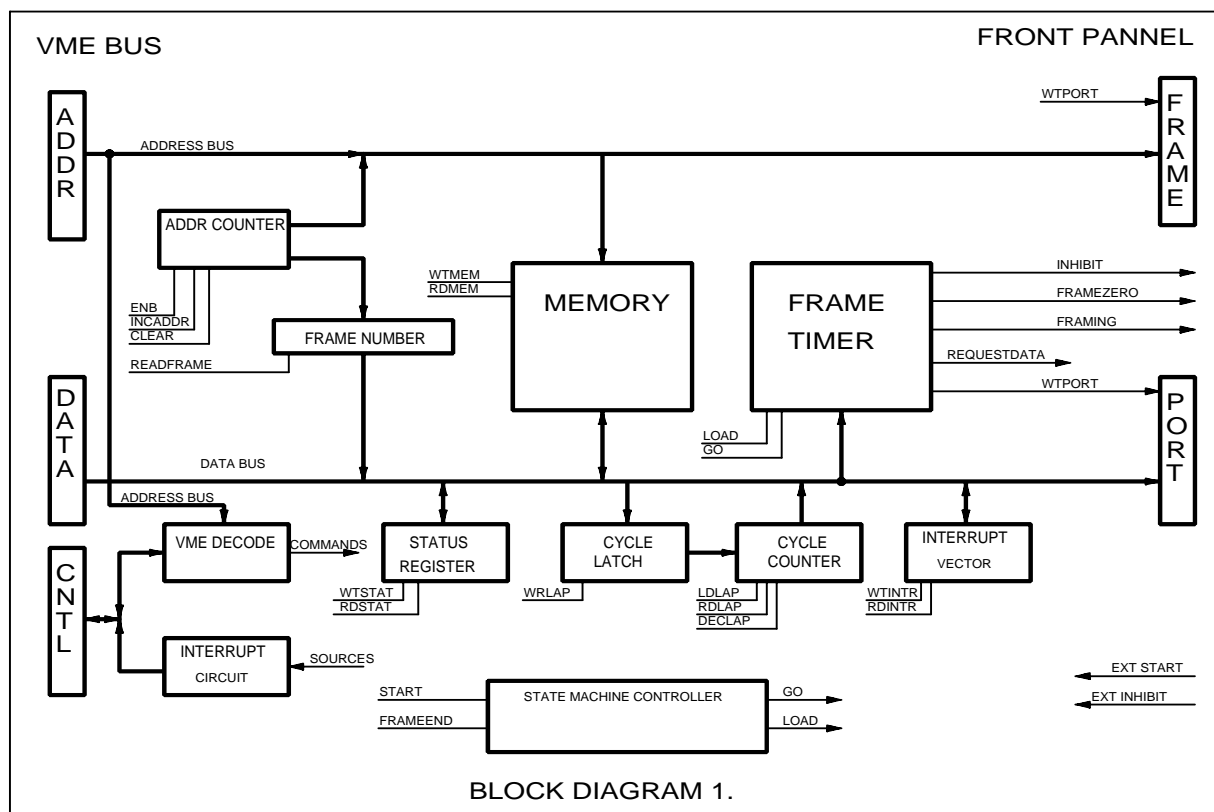
The address bus connects the VME port, address counter, memory and frame output port. Normally, the memory is addressed from the VME bus, allowing random access. When the TFG is running and the end of a frame is approaching, the address bus can be driven from the address counter. This enable the timing and port information to be read the memory and the frame output to be supplied.

The data bus connects the VME data bus, the internal registers, the memory, the frame timer and the output port (which drives the 8 front panel Lemos). Normally the data bus is available to allow VME read/write access to the memory and the control registers and counters. When the end of a frame is approaching, the control state machine uses the data bus to transfer the timing data to the frame timer and then the port data to the output port.

Block diagram 2 shows how data flows through the board when the module is running.

BLOCK DIAGRAM 2
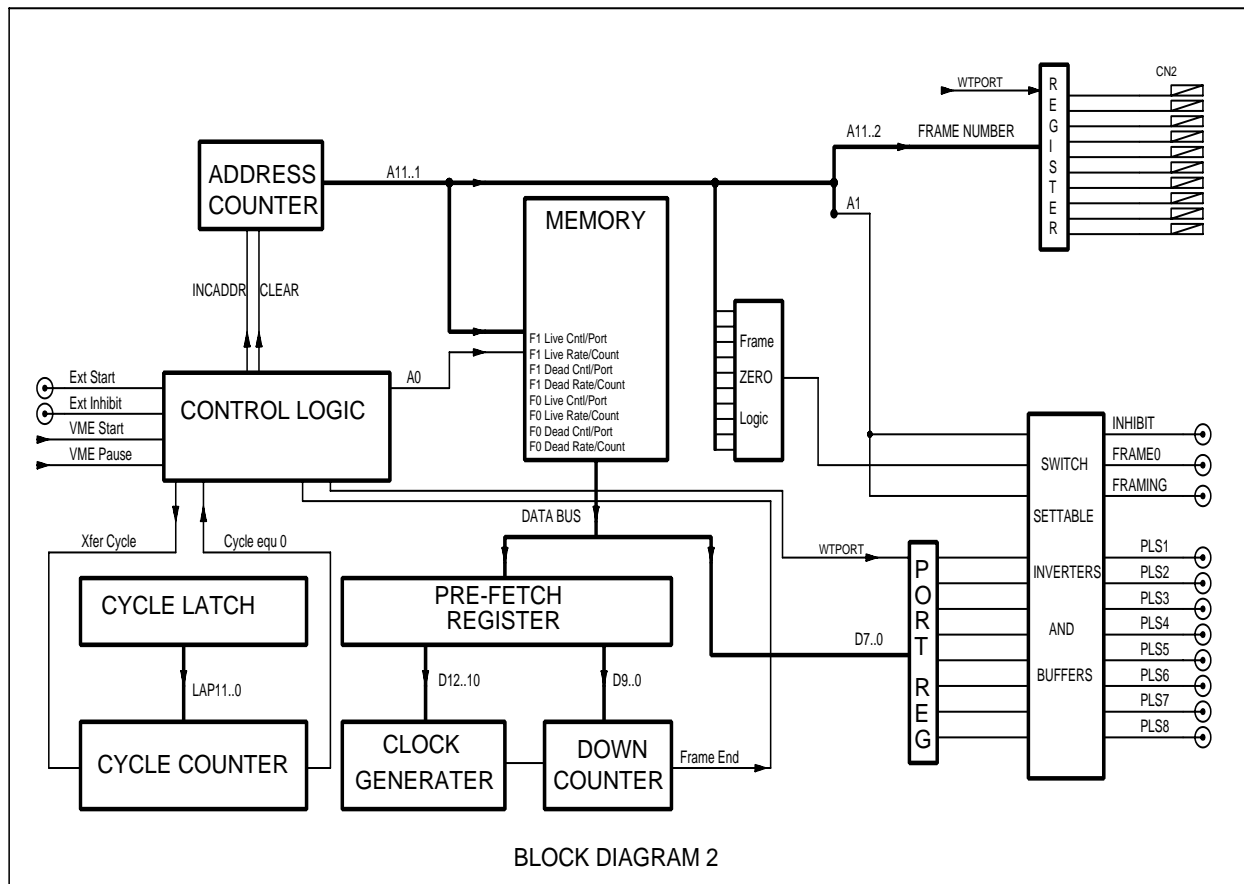
Before an experimental run, the memory is filled with the time framing information. This consists of a 13 bit rate + count word in one 16 bit memory location, followed by a 10 bit port data plus control word in the next 16 bit location. This is repeated for the live and dead part of each frame as described above. The number of cycles -1 required is written into the cycle latch. The status register can then be written to enable the external inputs and interrupts, as required. The module then waits for a VME start command or an external start, if enabled.

When the start signal is received, the number of cycles required is transferred from the cycle latch to the cycle counter. Simultaneously, the address counter is cleared and A0 is cleared so addressing the rate and count part of the first dead frame data. This timing information is stored in the pre-fetch register while the A0 is asserted to point to the control/port data in the memory. Then the timing rate information D12..10 is latched into the Clock generator, the timing count D9..0 is loaded into the down counter, the port data is written to the port register and the address counter is latched into the output register. Hence address counter A1 gives the INHIBIT signal and address counter A11..2 gives the current frame output -initially 0. The down counter then starts to counts down at a rate determined by the rate bits D12..10 supplied to the clock generator. At this point the address counter is incremented ready to ready to load the next frame. This is why the address counter appears one ahead of the current frame. Note also that the least significant word address

bit (A0) distinguishes between rate/count word and port/control word. Hence the layout of the frame counter on read - see above.

Four microseconds before the end of the frame, the frame end signal is asserted. This is used to warn the memory controller to finish any VME cycles and get the next frame data ready. Again the frame rate and count information is transferred to the pre-fetch register with A0 clear and then A0 is asserted to access the port/control data. At the exact end of the frame, the pre-fetched timing information is transferred to the clock generator and counter and A1 is used to determine the inhibit line. 100 ns later the port data and the frame number (address) are written. This guarantees approximately 100 ns frame address hold time after the change in inhibit marking the end of the live frame -(the frame address does not change between dead and live sections of the same frame). This hold time may be useful to systems storing data at the end of a live frame. If neither the pause or end of lap bit are set, then the counter is started and the next frame is timed, with no load time overhead. Otherwise some special processing occurs.

If the end of lap bit in the control word is set, then the cycle counter is checked. If it is zero then this is the last frame and when it ends the TFG returns to its idle state. In the idle state inhibit is asserted to disable data acquisition, marking the frame as dead. The frame number and port is reset to zero. Again address hold time is guaranteed between the frame address and the inhibit signal. If the cycle counter is not zero, then the address counter is reset and the cycle counter is decremented, ready for the next cycle. Hence the frame number will read zero during the last frame.

If the pause bit is set then the module pauses. The outputs have been set for this frame but the down counter is held. This means that inhibit will be asserted if the pause is in a dead frame, but negated if it is a live frame. The output port data will have been loaded from the memory. The module remains in this state until a VME or (if enabled) external start signal arrives. When the start signal arrives the counter proceeds to time the current frame. A pause bit can be set in any frame (including the first and last), but is more usually set in dead frames when no data is being collected.

The module allows dual porting of the memory and control/status registers. Except for the last 4 micro seconds of every frame, the VME bus is allowed access to the registers and memory. When the frame end signal is asserted as the end of the frame approaches any current VME cycles are finished - it is assumed within 3 µs, and then memory and busses are used to transfer the timing and port data. Any VME cycles occurring at this time will be delayed by up to 4 µs while this end of frame processing is completed.

Suggested Modes of Operation

Using the front panel outputs.

The Inhibit output is intended to enable and disable counting and hence is usually fed to the GATE or VETO input of the data acquisition system. The framing output is derived from the inhibit signal but may be of a different polarity. It is conventionally used to tell the data acquisition system to transfer data for this frame to memory (if appropriate). With the EC738 multi-channel scaler, the inhibit/veto signal can be combined with other veto conditions if required to block counting during live frames. The Framing signal will normally be connected directly to the LOAD input of the scaler, transferring the data at the end of the live frame.

The frame 0 output is asserted when during all of frame 0 (live and dead period). It can be used to trigger parts of the sample control system, or data acquisition systems which differentiate frame 0 from other frames.

The 8 general purpose outputs can be used to trigger parts of the sample control system or the data acquisition system. The internal port is set to zero during the reset or idle state. If the polarity control switches are all set to active high (off), then all the outputs will be low. If it is required to make any the outputs high during the reset state, then these signals are inverted using the associated on board switches. When the TFG is running or paused, the output bits are set according to the contents of the memory but subject to the same inversions as during the idle state. When the memory contains 0 the outputs are the same as the idle state.

Programming Conventions

For a purely time resolved experiment the TFG will be used with no pause bits. During dynamic processes of interest, is it usual to make the dead frames as short as possible (10us). However, when using the EC738 scaler, it is necessary to have dead frames to allow transfer of data from the scalers to the shadow resisters and then to the memory. With other systems the dead frames must be included to allow the address port of the TFG to be correctly aligned. If parts of a dynamic process are not of interest, then long dead frames can be used to disable counting during those periods. The first part frame of a frame pair is always the dead frame. This is a useful convention, because the first dead frame can be used to allow time for the dynamic process to be triggered by an output bit set at the beginning of the first dead frame. It is necessary to stick to this convention of dead frame

first to ensure operation with the EC738, as this determines address setup and hold times.

The TFG can also be used in experiments where the exact length of the live frames must be controlled, but the time between the live frames varies. I particular it is possible to allow CPU or manual intervention between frames. This might be to adjust sample conditions or to move a mono-chromator during experiments. Alternatively it may allow adjustment of the data acquisition system during automatic setup procedures.

In these cases the TFG could be setup with a pause bit in every dead frame. The dead frame length is again likely to be set to 10 μs. The TFG will then pause between each live frame allowing the CPU or manual. It then requires a `tfg_start()` or an external start to make it continue. The pause interrupt can be used to determine when the TFG has entered a pause state.

It is also possible to configure the TFG to perform bursts of time resolved collection separated by pauses which allow the experiment to reach certain detectable states. The pauses would be caused using the pause bit (normally in the dead frame) and a sensor on the experimental state would then end the pause by asserting external start. Again the dead frame will usually be set to 10 μs so there will be a 10 μs delay between the external start and the next live frame starting. If is necessary to collect data before the external start, or to measure the width of the pause, then it is possible to put the pause bit in a live frame. Data collection will continue during the pause so by supplying one of the scaler channels from a stable pulse generator, the pause width could be measured. Note also that is acceptable to use a mixture of external starts and VME starts. Hence such an experiment could be started using the VME start command and then continued by an external start.

Front Pannel Connectors.

There are two versions of TFG PCB. One provides the Time frame output on a 14 way IDC connector. The other uses a 25 way D-type connector with alternate wires grounded. The Inhibit and Framing or Veto and Transfer signals are also distributed in the 25 way D-type ribbon cable, removing the need for short LEMO cables accross the front of a bank of scalers.

14 way IDC version

Pin	Signal
1	Time frame Bit 0
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
11,12	Unused
13,14	Ground

25 way D-type version

Pin	Signal
1	Time frame Bit 0
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
11	VETO High = count Low = Stop
12	TRANSFER Transfer on rising edge.
13	Unused
14,25	Ground

Test program

A test program called `tfgtest` is supplied. This operates the board via the device driver and descriptor.

The test program can be started directly from the OS/9 command line by typing `tfgtest`. This will run the default set of tests on the board. Alternatively, the test program can operate an interactive manner. This is done using `tfgtest -i`. In this can the following screen will apear:

```
Time frame generator test options
0=>Device        tfg0   Can use device desciptor tfg0..3
1  Initialise    *      Tests that status register clears
2  Status        *      Tests read/writeable bits of status register
3  Memory        *      Tests read/write of memory
4  Polled Slow   *      Tests timing without interupts
5  Lap Counter   *      Writes lap, starts run and checks read of Lap
6  Output slow   *      Scans pulse outputs - watch LEDs toggle
7  Output fast          Exercises outputs - check timing with logic analyser
8  Single frames 1ms-1s Times single frames against internal clock
9  9ms frames    OS9    Time 9ms live frames using OS9 clock or external L.A
A  9ms with pause *     9ms live frames with pause in dead. Use L.A.
B  VME pause     *      Tests VME pause request
C  Stop          *      Prematurely stops tfg using init. Check outputs
D  ExtStart             Requires manual triggering or Ext Start
E  ExtInh               Requires manual triggering or Ext Inhibit
F  Messages      Normal Sets number of output messages
G  Loop                 Repeat selected test
H  Exit on err   *      Exit on first error/ try to continue


X  Exit program now             0..9 etc or UP/DOWN Select option
Y  Restore default options      [SPACE]       Toggle options
Z  Clear all options            [RETURN]      Run selected tests
```

Items marked with a * or a message wil be performed. Use the cursor keys or < and > or numbers 0..9 and A..H to move to the desired option. Then use space to toggle that option on/off. Finnally press RETURN to run the tests.

0	**Device**.	Sets device descriptor name.
1	**Initialise**.	Tests that the status register clears.
2	**Status**.	Tests that the writeable bits of the status register can be written and read back.
3	**Memory**	Tests read/write of memory using an incrementing pattern and then sliding ones patterns.
4	**Slow**	Tests timing without interrupts, using polled code.
5	**Lap Counter**	Tests loading of the lap or cycle counter. Since the cycle counter is loaded from a buffer latch, this writes to the cycle latch, starts a run and then reads back the cycle and checks it.
6	**Output slow**	This causes the outputs to toggle so a sliding one appears on the LEMO outputs. If Lemos with LEDs are fitted to the outputs, then the outputs can be checked by eye.
7	**Output fast**	This produces the same sliding one pattern as describe above, but much faster. The outputs can usefully be studied with a Logic analyser. This demonstrates the relative timing of the Inhibit, Framing, Frame 0, Frame number and LEMO Outputs.

8	**Single frames**	This sets up single time frames using 1ms through 100 s timing units. These frames are timed against the internal clock
9	**9ms frames**	This generates many time frames. All the live frames are of width 9 ms. The dead frames are of variable width. The total time taken for the runs can be measured against the internal OS9 clock. Additionally, the width of the live frames can be checked using a logic analyser.
A	**9ms with pause**	This generates many time frames. All the live frames are of width 9 ms. The dead frames all contain a pause bit. The widths of the 9ms live frames can be checked using a logic analyser.
B	**VME pause**	Tests VME pause request.
C	**Stop**	Prematurely stops TFG during a run using tfg_init. This checks that the tfg does stop. Additionally, it is possible to study the timing of the outputs using a logic analyser. In particular, address hold time is gauranteed after the assertion of Inhibit.
D	**ExtStart**	This tests the external start. The user is prompted do supply an edge to the ExtStart input. This can be done by shorting it to ground.
E	**ExtInh**	This tests the external inhibit. The user is prompted do supply an edge to the ExtInh input. This can be done by shorting it to ground.
F	**Messages**	The number of messages output can be set to Few, Normal or verbose.
G	**Loop**	Repeat selected test(s)
H	**Exit on err**	Exit on first error/ try to continue

The timing of the TFG against the internal OS/9 clock has limited accuracy. This is due to the time sharing of OS/9. A user process cannot control how long between an interrupt occurring and it processing the signal. Therefore the test program should be run on a system with no other processes running. Other processes can be sleeping or waiting, but not active. It is still normal for the program to reports errors of a few ticks. To test the End of Cycle interrupt, the test program causes 1024 EOC interrupts and counts them. These interrupts. If the operating system is busy, with interrupts disabled, some of these interrupts will be lost and the test will fail. This is accepted in the test program, but any user code should not attempt to count interrupts.

Device driver library

An OS/9 device driver is available. This is accessed via a C callable library. The functions available are describe below.

```
*********************************************************************
*                         tfg_disable ()                           *
*********************************************************************


SYNOPSIS:       #include "tfg.h"

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

FUNCTION:       Disable the external inputs to the time frame generator.
                Returns -1 on error & ERRNO for driver errors, else 0.



*********************************************************************
*                         tfg_enable ()                            *
*********************************************************************


SYNOPSIS:       #include "tfg.h"

                int tfg_enable (path, status)
                int path;                              /* Path number */
                int status;         /* Set status register bits 0,1,2 */

FUNCTION:       Write enable bits to the time frame generator status
                register (one or more bits OR'ed into the status).
                Returns -1 on error & ERRNO for driver errors, else 0.



*********************************************************************
*                         tfg_init ()                              *
*********************************************************************


SYNOPSIS:       #include "tfg.h"

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

FUNCTION:       Initialise the time frame generator.
                Returns -1 on error & ERRNO for driver errors, else 0.



*********************************************************************
*                         tfg_irqdis ()                            *
*********************************************************************


SYNOPSIS:       #include "tfg.h"

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

FUNCTION:       Disable interrupts in the time frame generator.
                Returns -1 on error & ERRNO for driver errors, else 0.
```

```
***********************************************************************
*                       tfg_irqenb ()                                 *
***********************************************************************


SYNOPSIS:        #include "tfg.h"

                 int tfg_irqenb (path, status, procid, signum)
                 int path;                            /* Path number */
                 int status;                  /* Status register bits */
                 int procid;                  /* Process id to signal */
                 int signum;                       /* Signal to send */

FUNCTION:        Write interrupt enable bits to the time frame generator
                 status register (one or more bits OR'ed into the
                 status register). Returns -1 on error & ERRNO for
                 driver errors, else 0.



***********************************************************************
*                       tfg_pause ()                                  *
***********************************************************************


SYNOPSIS:        #include "tfg.h"

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

FUNCTION:        Pause the time frame generator. Returns -1 on error &
                 ERRNO for driver errors, else 0.



***********************************************************************
*                       tfg_prcsig ()                                 *
***********************************************************************


SYNOPSIS:        #include "tfg.h"

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

FUNCTION:        Inquire which process the time frame generator signals
                 Returns -1 on error & ERRNO for driver errors, else
                 process ID



***********************************************************************
*                       tfg_rdframe ()                                *
***********************************************************************


SYNOPSIS:        #include "tfg.h"

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

FUNCTION:        Read the time frame generator (EC740) current frame
                 Returns -1 on error & ERRNO for driver errors, else
                 current frame.
```

```
*********************************************************************
*                         tfg_rdmem ()                             *
*********************************************************************


SYNOPSIS:       #include "tfg.h"

                int tfg_rdmem (path, npts, buff)
                int path;                              /* Path number */
                int npts;         /* Nos memory locations <1 - 4096> */
                short *buff;                          /* Users buffer */

FUNCTION:       Read the time frame generator memory. Returns -1 on
                error & ERRNO for driver errors, else 0



*********************************************************************
*                         tfg_rdstat ()                            *
*********************************************************************


SYNOPSIS:       #include "tfg.h"

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

FUNCTION:       Read the time frame generator status register. Returns -1
                on error & ERRNO for driver errors, else current status.



*********************************************************************
*                         tfg_start ()                             *
*********************************************************************


SYNOPSIS:       #include "tfg.h"

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

FUNCTION:       Start time frame generator framing. Returns -1 on error
                & ERRNO for driver errors, else 0.



*********************************************************************
*                         tfg_wrcycle ()                           *
*********************************************************************


SYNOPSIS:       #include "tfg.h"

                int tfg_wrcycle (path, cycles)
                int path;                              /* Path number */
                int cycles; /* No of TFG cycles to perform <0 - 4095> */

FUNCTION:       Write to the time frame generator cycle counter.
                Returns -1 on error & ERRNO for driver errors, else
                current frame.
```

```
**********************************************************************
*                          tfg_wrmem ()                              *
**********************************************************************


SYNOPSIS:       #include "tfg.h"

                int tfg_wrmem (path, npts, buff)
                int   path;                            /* Path number */
                int   npts;        /* Nos memory locations <1 - 4096> */
                short *buff;                           /* Users buffer */


FUNCTION:       Write to the time frame generator memory. Returns -1 on
                error & ERRNO for driver errors, else current frame.
```

Use of TFG and Scaler together

Cabling

It is recommended that these two modules are always in the same VME crate - which will be the usual case.

The TFG address port is connected to the scaler address port using a 14 way IDC cable.

The TFG INHIBIT output is connected to the scaler veto input.

The TFG Framing output is connected to the scaler load input.

Switches

The TFG switches are set as follows:-

        E23-3 ON

        E23-5 OFF   VETO is active low.

The rest of the TFG switches can be set according to the use of external pause and start and any pulse outputs.

The scaler switch E13 is set to TTL (left), to use TTL VETO and LOAD signals

Programming example

A demonstration program demo.c is supplied with the scaler and TFG. This program uses the Daresbury device drivers and descriptors, which must be loaded before the program is run.

The program asks the user for the live time length in seconds, the number of frames and the number of cycles. All the live frames are set to this length. The dead frames are set to the minimum. Once the last number is entered the program runs the TFG and scaler collecting the required number of frames. At the end of the run the data is written to the file scaler.dat as ASCII numbers.

Example program

```c
/***************************************************************************
*
*   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);
}
```

Circuit Diagram

The Circuit Diagram can be found in the appendix. It consists of six basic sections, these are the VME interface, the Memory, the IRQ driver circuit, the Frame and Port output latches, the Module Control (MAX 1) and the timing generator (MAX 2).