Device and Driver Support for F3RP61 (Ver. 1.1.1)

J. Odagiri

Precautions:

- **The device / driver support requires BSP R2.01 of F3RP61 (e-RT3 2.0 / Linux) to build it on a host computer and to execute it on the F3RP61 CPU.**
- **The device / driver support relies on a user-level library, libm3.so.1.0.0, as well as the kernel-level driver module, m3iodrv.ko (m3iodrvRT.ko for the kernel CONFIG_PREEMPT_RT patch applied) included in the BSP at run-time.**

Contents

Main Change from the previous version (Ver. 1.1.0)

- Makefile undr src directory was modified so that it builds the library, libf3rp61.a, only if T_A = linux-f3rp61.

Main changes from the original version (Rev. 1.0)

- Mode register access support (See, section 3.4)
- New interface for communication with Sequence CPU (See, section 6.1.1)
- FL-net support (See, section 8)

1. Overview

This device / driver support can be used to run EPICS iocCore on an embedded Linux controller, F3RP61 (e-RT3 2.0/Linux), made by Yokogawa Electric Corporation. The controller, F3RP61, can access most of the I/O modules of the FA-M3 PLC on the PLC-bus. This feature opens way for making an FA-M3 PLC itself a new type of IOC. The device / driver support provides interfaces for iocCore to access I/O modules as well as ordinary sequence CPUs that works on the PLC-bus. The device / driver support is implemented by wrapping the APIs of the kernel-level driver and the user level library, which are included in the Board Support Package (BSP) R2.01 of F3RP61.

As to I/O modules, the device / driver support offers only primitive method to access the relays and registers. In the sense that any relays and registers of a supported I/O module can be accessed by using the device / driver support, it is universal. However, in order to handle a special module that requires some sequence logic to execute I/O operation, such as motion control module, the sequence logic needs to be implemented by using an EPICS sequencer program by the user. (The sequence logic to handle a special module is implemented by using a ladder program when a sequence CPU is used to execute the I/O operation. The EPICS sequencer program is used to replace the ladder program. See section 4 for more detail.) If some initialization is required on a special module, it can be done by using an EPICS sequencer program, or a runtime database comprised of records that have the PINI field value of "YES".

An F3RP61 can work as an IOC with / without sequence CPUs that run ladder programs. If no sequence CPU is on the PLC-bus, the F3RP61 should manage all the I/O activities. If one or more sequence CPUs are on the PLC-bus, some of the I/O modules can be used with the sequence CPUs and the others can be used with the F3RP61. It is recommended that the I/O module under the sequence CPU's control be accessed indirectly by the IOC (F3RP61) through the internal devices of the sequence CPUs. (See section 5 for more detail.)

Digital input modules of FA-M3 can interrupt CPUs upon a change of the state of the input signal. The BSP of F3RP61 has a function that transforms the interrupt into a message to a user-level process running on it. Based on this function, the device / driver support supports processing records upon an I/O interrupt.

2. Device Type

In order to use the device / driver support, the device type (DTYP) field of the record must be set to either “F3RP61” or “F3RP61Seq”. The former is for accessing the relays and registers of I/O modules and a shared memory. The latter is for accessing internal devices (“D”, “I”, “B”) of the sequence CPUs on the same base unit.

3. Accessing I/O Module

3.1. Accessing Input Relay (X)

Input relays are read-only devices. Binary input (bi) records, multi-binary input direct (mbbiDirect) records, long input (longin) records and analog input (ai) records are supported by the device support. The three numbers, a unit number, a slot number and an input relay number must be specified in the INP field as the following example shows.

```
record(bi, “f3rp61_example_1”) {
        field(DTYP, “F3RP61”)
        field(INP, “@U0,S2,X1”)
 }
```

The above record reads a value of either “on” (1) or “off” (0) from the first input relay (X1) of an I/O module in the second slot (S2) of the main unit (U0). The next example

shows how to read 16 bits of status data on a group of input relays by using an mbbiDirect record.

```
record(mbbiDirect, "f3rp61_example_2") {
        field(DTYP, "F3RP61")
        field(INP, "@U0,S2,X1")
}
```

In this case, the number "1" as in "X1" specifies the first relay to read. The status bits on the 16 relays, X1, X2, …, X16 are read into the B0, B1, …, BF fields of the mbbiDirect record.

The next example shows how to read a digital value of 16 bits on a group of input relays by using longin records.

```
record(longin, "f3rp61_example_3") {
        field(DTYP, "F3RP61")
        field(INP, "@U0,S2,X1")
}
```

In this case, it is assumed that the Least Significant Bit (LSB) is on X1 and the Most Significant Bit (MSB) is on X16, and the digital value on the input relays is assumed to be signed short. If the value is unsigned short, "&U" must follow the relay number as follows.

```
record(longin, "f3rp61_example_4") {
        field(DTYP, "F3RP61")
        field(INP, "@U0,S2,X1&U")
}
```

The rules explained in the last two examples apply to the ai record. The value read from the input relays goes into the raw value (RVAL) field of the ai record.

3.2. Accessing Output Relay (Y)

Output relays are read-write devices. Just replacing 'X' with 'Y' in the INP fields of

the example records shown in the previous subsection suffices to read output relays.

In order to write output relays, binary output (bo) records, multi-bit binary output direct (mbboDirect) records, long output (longout) records, and analog output (ao) records can be used. The following example shows how to write an output relay by using a bo record.

```
record(bo, "f3rp61_example_5") {
        field(DTYP, "F3RP61")
        field(OUT, "@U0,S2,Y1")
}
```

The above record writes a value of either "on" (1) or "off" (0) onto the first output relay (Y1) of an I/O module in the second slot (S2) of the main unit (U0).

The next example shows how to write 16 bits of data onto a group of output relays by using an mbboDirect record.

```
record(mbboDirect, "f3rp61_example_6") {
        field(DTYP, "F3RP61")
        field(OUT, "@U0,S2,Y1")
}
```

In this case, the number in "Y1" specifies the first relay to write. The 16-bits data of the mbboDirect record, B0, B1, …, BF, are written onto the output relays, Y1, Y2, …, Y16 respectively.

The next example shows how to write a value of 16 bits onto a group of output relays by using a longout record.

```
record(longout, "f3rp61_example_7") {
        field(DTYP, "F3RP61")
        field(OUT, "@U0,S2,Y1")
}
```

In this case, it is assumed that the Least Significant Bit (LSB) goes onto Y1 and the Most Significant Bit (MSB) goes onto Y16, and the value is signed short. If the value is unsigned short, "&U" must follow the relay number as follows.

```
record(longout, “f3rp61_example_8”) {
        field(DTYP, “F3RP61”)
        field(OUT, “@U0,S2,Y1&U”)
}
```

The rules explained in the last two examples apply to the ao record. The value in the raw value (RVAL) field of the ao record is written onto the output relays.

3.3 Accessing Data Register

Analog I/O modules and other special modules, such as motion control modules, etc., have many registers to hold the I/O data and relevant parameters. In order to read / write these registers, longin / longout, ai / ao and mbbiDirect / mbboDirect records are supported. While some of the registers hold 32-bit data, the device / driver support supports only reading / writing a value of 16 bits. 32-bit registers must be read / written, according to the specification of the I/O module, by using two records, one for the upper half and the other for the lower half of the 32 bits of the value.

The following example shows how to use a longin record to read a 16-bit register.

```
record(longin, “f3rp61_example_9”) {
        field(DTYP, “F3RP61”)
        field(INP, “@U0,S3,A1”)
}
```

The above record reads a value of 16-bit data from the first register (A1) of a module in the third slot (S3) of the main unit (U0).

The following example shows how to use a longout record to write a 16-bit register.

```
record(longout, “f3rp61_example_10”) {
        field(DTYP, “F3RP61”)
        field(OUT, “@U0,S3,A1”)
}
```

The above record writes a value of 16-bit data onto the first register (A1) of a module in the third slot (S3) of the main unit (U0).

The rules in the above two examples apply to ai /ao, mbbiDirect / mbboDirect records. The value read from (written onto) the device comes in (goes out) via the RVAL field of the records.

As to longin and ai type records, the "&U" option can be specified at the end of the INP field to read unsigned 16-bit data as shown below.

```
record(longin, "f3rp61_example_11") {
        field(DTYP, "F3RP61")
        field(INP, "@U0,S3,A1&U")
}
```

3.4 Accessing Mode Register

Non-intelligent digital I/O modules have mode registers to hold the scan rates, interrupt raising conditions (rising edge / falling edge), filtering conditions of the I/O channels. In order to access the mode registers, an mbbiDirect /mbboDirect record can be used with a character for addressing, 'M', in the INP / OUT field of the records.

The following example shows how to use an mbbiDirect record to read a 16-bit register.

```
record(mbbiDirect, "f3rp61_example_12") {
        field(DTYP, "F3RP61")
        field(INP, "@U0,S3,M1")
}
```

The above record reads a value of 16-bit data from the first mode register (M1) of a non-intelligent digital I/O module in the third slot (S3) of the main unit (U0).

The following example shows how to use an mbboDirect record to write a 16-bit register.

```
record(mbboDirect, "f3rp61_example_13") {
        field(DTYP, "F3RP61")
        field(OUT, "@U0,S3,M1")
}
```

The above record writes a value of 16-bit data into the first mode register (M1) of a non-intelligent digital I/O module in the third slot (S3) of the main unit (U0). For example, if the I/O module is a digital input module and if you set the B0 field of the above record, it results in specifying that a part of the I/O channels (from X25 to X32) of the module raise interrupts upon the falling edge of the input signals. For more details, please consult relevant hardware manuals from Yokogawa Electric Corporation.

Note that the mbboRecord always outputs a value of zero when the record gets processed by being written a value into its VAL field regardless of whatever the value is. The author is not sure if the behavior is just a specification or a bug of the mbboRecord. At any rate, if you complete setting the conditions on a digital I/O module bit by bit by using B0, B1, B2, …, BF fields of an mbboRecord, you are free from the problem mentioned above.

4. Handling Special Module

This section describes how to handle special modules that require some sequence logic to execute I/O operations. As an example, we consider a motion control module that controls the motion of a stepping motor by sending a train of pulses to the motor driver.

Suppose you drive a motor dedicated to an axis. In the first place, you make the F3RP61 set the number of pulses (distance to move) into a register of the motion control module by using longout or ao record(s). (While the parameter is 32 bit long, the motion control modules do not support long word (32 bits) read / write operation. Two records, therefore, are necessary to set the upper 16 bits and the lower 16 bits.)

Next, you make the F3RP61 turn on an output relay (EXE) to trigger the action. This can be performed by putting the value of one (1) into the VAL field of the following

record. (The relay number varies from type to type. The following example records are just for illustration. See the manual of the module you use for more detailed information. We assume the motion control module is in slot 4.)

```
record(bo, "f3rp61_motion_exe") {
        field(DTYP, "F3RP61")
        field(OUT, "@U0,S4,Y33")
 }
```

The motion control module will respond to the EXE command by turning on an input relay (ACK) if no error is found in the parameters given. The F3RP61 needs to check the ACK by using a record shown below.

```
record(bi, "f3rp61_motion_ack") {
        field(SCAN, ".1 second")
        field(DTYP, "F3RP61")
        field(INP, "@U0,S4,X1")
 }
```

And then, the operation (sending pulses) starts. The F3RP61 is required to turn off the EXE after the ACK is turned on. The motion control module, then, turns off the ACK after the EXE is turned off.

When the operation (sending pulses) completes, the motion control module informs the F3RP61 of the completion by turning on yet another input relay (FIN). Just like the ACK, the FIN needs to be checked by using a record shown below.

```
record(bi, "f3rp61_motion_fine") {
        field(SCAN, ".1 second")
        field(DTYP, "F3RP61")
        field(INP, "@U0,S4,X5")
 }
```

Note that the records to monitor the input relays must be processed periodically. You might want to choose to use the interrupt support explained in section 7 instead of periodic scanning, though the author has not yet tried it in using motion control modules.

The essential part of the EPICS sequencer program to manage the sequence described above in words will look like as follows.

```
int   exe;
in   ack;
int   fin;
assign   exe   f3rp61_motion_exe;
assign   ack   f3rp61_motion_ack;
assign   fin   f3rp61_motion_fin;
monitor   exe;
monitor   ack;
monitor   fin;

ss exe_move {
        state wait_exe {
                when (exe) {
                } state wait_ack
        }
        state wait_ack {
                when (ack) {
                        exe = FALSE;
                        pvPut(exe);
                } state_wait_fin
        }
        state wait_fin {
                when(fin) {
                } state wait_exe
        }
}
```

The author reminds you that the above example is just for illustration. The sequencer program for the real operation involves a little bit more to handle other types of commands for the motion control module, and to handle exceptions that can occur in the sequence (for example, an error caused by a wrong parameter set by the user).

5. Cautions in Using F3RP61 and Sequence CPU Side-by-side

This section gives you important cautions in using an F3RP61 CPU and a normal sequence CPU on the same unit.

In many cases, this type of multi-CPU configuration is used in case you use a normal sequence CPU to implement an interlock logic that is required high reliability. In that case, the normal sequence CPU dedicates to handle the interlock signals while the F3RP61 CPU is used to take care of other control, or just to monitor the status of the interlock signals. In this case, you need to be cautious of the following two points.

- The sequence CPU must be in the first slot (slot 1) because the CPU in the first slot becomes the master of the unit and the muster resets the whole system upon rebooting.
- The F3RP61 CPU should NOT have an access, regardless of read or write, to the I/O modules that are used with the normal sequence CPU for the interlock.

The reason of the second point is as follows. If an I/O module is accessed by the F3RP61 CPU, the I/O module recognizes and remembers that the F3RP61 CPU is one of its masters. When the Linux system on the F3RP61 CPU is rebooted, the F3RP61 CPU broadcasts the fact by using a signal on the PLC-bus. The I/O modules that have been accessed by the F3RP61 CPU and recognize it as one of their masters reset themselves when they detect the signal. It makes the I/O modules un-accessible by the sequence CPU and makes the ladder program stop with I/O errors. For this reason, it is highly recommended that you make the F3RP61 CPU read the status of interlock indirectly via some internal devices ('I', 'D', 'B') of the sequence CPU or, through the shared memory ('E', 'R') by using a method described in the next section.

On the other hand, if one or more I/O modules are used with an F3RP61 CPU for some control in the multi-CPU configuration, you need to tell the sequence CPU not to touch the I/O modules under the F3RP61 CPU's control. This setting can be done on the sequence CPU by using the ladder development software, WideField2 (or WideField3). From the menu of your "Project", select "Configuration" and then, select "DIO Setup". Change the default setup from "Use" to "Not used" for the I/O modules. Otherwise, the sequence CPU overwrites the data of the output channels of the modules with the value of zero even when any I/O execution commands on the I/O modules do not appear explicitly in the ladder program.

6. Communication with Sequence CPU

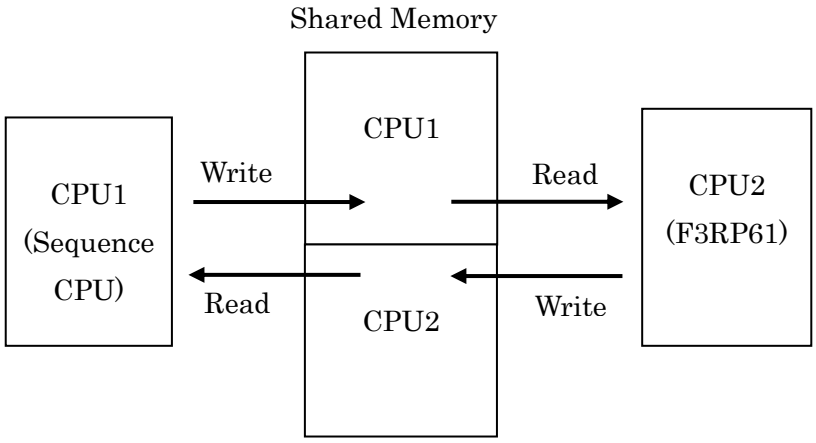
Two different types of methods are supported for an F3RP61 to communicate with the sequence CPUs that work on the same base unit. One is shared-memory-based communication and the other is message-based communication. The former is fast access that finishes instantly, just like the access to the I/O relays and registers of an I/O module. The latter is slow access that takes a few milliseconds of time to complete. For this reason, two different DTYPs are defined in the device / driver support. The DTYP field needs to be set to "F3RP61" for the former (synchronous) and "F3RP61Seq" for the latter (asynchronous).

6.1. Communication Based on Shared Memory

The following is the basics to understand how the communication between F3RP61 CPUs and normal sequence CPUs works.

- Each CPU, a sequence CPU or an F3RP61 CPU can have regions allocated to it.
- Any CPU, a sequence CPU or an F3RP61 CPU can write into only the regions allocated to it.
- Any CPU, a sequence CPU or an F3RP61 CPU can read out from any regions.

In order to make the story simple, we consider a case where only one sequence CPU in slot 1 works with only one F3RP61 CPU in slot 2 on the same base unit. From the rules mentioned above, how we use the shared memory to make the two CPUs communicate with each other is clear. If the data go from the sequence CPU (CPU1) to the F3RP61 CPU (CPU2), use the area allocated to the sequence CPU (CPU1), and vice versa, as shown in the figure below.



6.1.1. Communication Based on Shared Memory Using New Interface

In BSP R2.01 of F3RP61, a set of new APIs are newly supported to access the shared memory (shared relays and shared registers). The device / driver support (Ver. 1.1.0 or later) supports accessing the shared memory based on the new APIs. In this case, users need to put the following IOC command in the startup script for an F3RP61-based IOC to specify how many shared relays and shared registers are allocated to each of the CPUs.

```
f3rp61ComDeviceConfigure(0, 512, 256, 64, 32)
f3rp61ComDeviceConfigure(1, 512, 256, 64, 32)
```

The command needs to be executed prior to the call to iocInit(). The first line implies that 512 bits of shared relays, 256 words of shared registers, 64 bits of extended shared relays and 32 words of extended shared registers are allocated to CPU1(0 + 1), say, a normal sequence CPU in slot 1. The next line means that the same numbers of shared relays and shared registers are allocated to CPU2(1 + 1), say, an F3RP61 CPU in slot 2.

Note that the users themselves are responsible for making the configuration done on the F3RP61-side as shown above consistent with the configuration done on the sequence CPU-side by using WideField2 (or WideField3).

The following example shows how to read a shared relay by using a bi record.

```
record(bi, "f3rp61_example_14") {
        field(DTYP, "F3RP61")
        field(INP, "@E1")
 }
```

The bi record can be used to read the first the first shared relay (E1). In this case, the relay (E1) can be allocated to any CPU as mentioned earlier.

The following example shows how to write a shared relay by using a bo record.

```
record(bo, "f3rp61_example_15") {
        field(DTYP, "F3RP61")
        field(OUT, "@E1")
 }
```

The bo record can be used to write the first the first shared relay (E1). In this case, the relay (E1) must be allocated to the F3RP61-based IOC as mentioned earlier.

The following example shows how to read a shared register by using a longin record.

```
record(longin, “f3rp61_example_16”) {
        field(DTYP, “F3RP61”)
        field(INP, “@R1”)
 }
```

The longin record can be used to read the first the first shared register (R1). In this case, the register (R1) can be allocated to any CPU.

The following example shows how to write a shared register by using a longout record.

```
record(longout, “f3rp61_example_17”) {
        field(DTYP, “F3RP61”)
        field(OUT, “@R1”)
 }
```

The longout record can be used to write the first the first shared register (R1). In this case, the register (R1) must be allocated to the F3RP61-based IOC.

Ai / ao, mbbiDirect /mbboDirect are also supported to read / write the shared registers.

6.1.2. Communication Based on Shared Memory Using Old Interface

The author recommends that you choose the method to access the shared memory (shared relays and shared registers) based on the new APIs since it is much easier to understand. The device /driver support, however, still supports accessing the shared memory based on the old APIs for backward compatibility. If you choose this option, you need to know the following points.

- While a sequence CPU can access shared relays bit by bit, an F3RP61 CPU can access the shared relays only by word.
- The specification of the INP / OUT fields is different from that explained in 6.1.1.
- You need to understand the address map shown below.

In order to make the story simple, we consider a case where we have only two CPUs, a sequence CPU in slot 1 (CPU1) and an F3RP61 CPU in slot 2 (CPU2). In addition, we assume that we allocate 512 bits (32 words) of shared relays and 256 words of shared

registers to both CPU1 (a sequence CPU) and CPU2 (an F3RP61 CPU). In this case, the F3RP61 gets to see the following flat memory space, of which address starts with zero (the first “R00000” of the left most column).

```
---------------------------------------------------------------------------------------
R00000: E00001 – E00016          Allocated to CPU1 (sequence CPU)
R00001: E00017 – E00032
R00002: E00033 – E00048
            …
R00031: E00497 – E00512
---------------------------------------------------------------------------------------
R00032: E00513 – E00528          Allocated to CPU2 (F3RP61 CPU)
R00033: E00529 – E00544
R00034: E00545 – E00560
            …
R00063: E00497 – E01024
---------------------------------------------------------------------------------------
R00064: R00001                   Allocated to CPU1 (sequence CPU)
R00065: R00002
R00066: R00003
            …
R00319: R00256
---------------------------------------------------------------------------------------
R00320: R00257                   Allocated to CPU2 (F3RP61 CPU)
R00321: R00258
R00322: R00259
            …
R00575: R00512
---------------------------------------------------------------------------------------
```

The following record allows the F3RP61 CPU to read the first 16 bits of shared relays (E00001 –E00016) all at once.

```
record(mbbiDirect, “f3rp61_example_18”) {
```

```
        field(DTYP, "F3RP61")
        field(INP, "@CPU1,R0")
}
```

Note that, in this case, the address you need to specify in the INP field is the left most number in the mapping table shown above. The bi record is not supported to read a shared relay because CPU2 (F3RP61 CPU) can access shared relays only by word when you choose the old-API-based access.

Similarly, the first shared register allocated to CPU1 (sequence CPU) can be read by using the following record.

```
record(mbbiDirect, "f3rp61_example_19") {
        field(DTYP, "F3RP61")
        field(INP, "@CPU1,R64")
}
```

Both R0 and R64 in the last two examples cannot be written by the CPU2 (F3RP61 CPU) since they are allocated to CPU1 (sequence CPU).

Next, we consider CPU2 (F3RP61 CPU) writing a value into a region allocated to it.

```
record(mbboDirect, "f3rp61_example_20") {
        field(DTYP, "F3RP61")
        field(OUT, "@CPU2,R32")
}
```

On the CPU1 (sequence CPU)-side, it gets to see a write onto 16 bits of shared relays (E00513 – E00528) occur all at once upon the above record processing.

Similarly, the first shared register allocated to CPU2 (F3RP61 CPU) can be written by using the following record.

```
record(mbboDirect, "f3rp61_example_21") {
        field(DTYP, "F3RP61")
        field(OUT, "@CPU2,R320")
}
```

Needless to say, CPU2 (F3RP61 CPU) can read back the data written from both R32

(E00513 – E00528) and R320 (R00257) by using an appropriate input type record.

You might think that there is no reason for specifying the CPU-numbers in the INP fields of the examples shown above since the F3RP61 CPU sees the shared memory as a single flat memory space starting with the first address of zero. However, we do need to specify the CPU-numbers as shown in the examples. We do not discuss it any more since it is a bit complicated story.

Ai / ao and longin / longout, are also supported to read / write the shared registers.

6.2. Accessing Internal Device of Sequence CPU

The alternative method for an F3RP61 to communicate with a sequence CPU is to use the message-based transaction. The following example shows how to set (1)/ reset (0) an internal relay ('I'), say, 'I4', of CPU1 (a sequence CPU in slot 1).

```
record(bo, "f3rp61_example_22") {
        field(DTYP, "F3RP61Seq")
        field(OUT, "@CPU1,I4")
}
```

Note that the device type must be "F3RP61Seq" in this case. In order to read back the result, you can use the following record.

```
record(bi, "f3rp61_example_23") {
        field(DTYP, "F3RP61Seq")
        field(INP, "@CPU1,I4")
}
```

In order to write a data register ('D'), say, 'D7', of CPU1 (a sequence CPU in slot 1), the following record can be used.

```
record(longout, "f3rp61_example_24") {
        field(DTYP, "F3RP61Seq")
        field(OUT, "@CPU1,D7")
}
```

Again, the result can be read back by using the following record.

```
record(longin, "f3rp61_example_25") {
        field(DTYP, "F3RP61Seq")
        field(INP, "@CPU1,D7")
}
```

Ai/ao and mbbiDirect/mbboDirect are also supported to read the data registers ("D"). Another internal device, file registers ('B'), can be read / written by using those record types.

Accessing internal devices other than 'I', 'D' and 'B' is not currently supported.

7. I/O Interrupt Support

Digital input modules of FA-M3 can interrupt the F3RP61-based IOC when they detect a rising edge or falling edge of the input signals. The kernel-level driver of the BSP can transform the interrupt into a message to a user-level process. Based on the function, processing records by I/O interrupt is supported with the device / driver support. Any records that have the DTYP field value of "F3RP61" and the SCAN value of "I/O Intr" get processed upon an interrupt on a specified channel of the specified module. This feature allows you to trigger a read / write operation by external trigger signals.

Suppose a unit which comprises of an F3RP61 in slot 1, a digital input module in slot2, and an A/D module in slot 3. The following record reads the first data register (A1) of the A/D module upon a trigger input into the first channel (X1) of the digital input module. (The INP field format takes the form of "@I/O_data_channel:interrupt_source".)

```
record(ai, “f3rp61_example_26”) {
        field(DTYP, “F3RP61”)
        field(SCAN, “I/O Intr”)
        field(INP, “@U0,S3,A1:U0,S2,X1”)
}
```

If you have a D/A module in slot 4, in addition to the module configuration mentioned above, you can write an output value into the first data register (A1) of the D/A module upon the same trigger input by using the following record.

```
record(ao, “f3rp61_example_27”) {
        field(DTYP, “F3RP61”)
        field(SCAN, “I/O Intr”)
        field(OUT, “@U0,S4,A1:U0,S2,X1”)
}
```

The following example might seem a little bit strange, but it helps you see how quick an F3RP61-baed IOC can respond to interrupts.

```
record(bi, “f3rp61_example_28”) {
        field(DTYP, “F3RP61”)
        field(SCAN, “I/O Intr”)
        field(INP, “@U0,S2,X1:U0,S2,X1”)
}
```

The bi record reads the status (level) of the relay for the trigger input. Suppose the trigger signal is a pulse and the digital module generates an interrupt at the rising edge. If the bi record reads the status before the signal level falls down, the record reads the status of “on” (1). Otherwise, it reads the status of “off” (0). By changing the pulse duration with checking the record value, you can roughly measure the time required for the record to get processed with the rising edge of the trigger signal as the starting point.

8. FL-net Support

There are two different methods in using FL-net. One is based on message transmission and the other is based on cyclic transmission. The device / driver support supports only the latter with fixed link refresh period of 10 milliseconds. It does not support FL-net in multi-CPU configuration at present. (setM3FlnSysNo() is called without specifying sysNo in the driver support.)

In order to use FL-net with an F3RP61-based IOC, users need to put the following IOC command in the startup script for an F3RP61-based IOC to specify how many link relays and link registers are allocated to each of the links.

```
f3rp61LinkDeviceConfigure(0, 512, 256)
f3rp61LinkDeviceConfigure(1, 512, 256)
```

The command needs to be executed prior to the call to iocInit(). The first line implies that 512 bits of link relays and 256 words of link registers are allocated to Link1(0 + 1). The next line means that the same numbers of link relays and link registers are allocated to Link2(1 + 1). (An F3RP61 CPU can handle up to two FL-net interface modules, i.e., up to two links though the author have tested only one link so far.)

Allocation of the link relays and link registers to each of the nodes on a link needs to be done on a sequence CPU-side by using WideField2 (or WideField3). There is nothing to be done on the F3RP61-based IOC on this point. Only allocation of the link relays and link registers to each of the links is needed by using the command described above.

How to make an F3RP61-based IOC communicate with other nodes on a link is similar with how to make the F3RP61-based IOC communicate with another CPU on the same base unit through shared memory based on the new APIs as described in 6.1.1.

The following example shows how to read a link relay.

```
record(bi, "f3rp61_example_29") {
        field(DTYP, "F3RP61")
        field(INP, "@L00001")
 }
```

The bi record can be used to read the first link relay allocated to a node on the Link1(0 + 1). The first zero of "L00001" subsequent to the leading "L" specifies the link1(0 + 1) and

the trailing “0001” specifies the address of the link relay on the link. (The first link relay of the Link2(1 + 1) can be addressed by “L10001”.) In this case, since it is a read access, the link relay can be allocated to any of the nodes on the link.

The following example shows how to write a link relay.

```
record(bo, “f3rp61_example_30”) {
        field(DTYP, “F3RP61”)
        field(OUT, “@L00001”)
 }
```

The bo record can be used to write the first link relay allocated to a node on the Link1(0 + 1). In this case, since it is a write access, the relay must be allocated to the F3RP61-based IOC as a node on the link.

The following example shows how to read a link register.

```
record(longin, “f3rp61_example_31”) {
        field(DTYP, “F3RP61”)
        field(INP, “@W00001”)
 }
```

The longin record can be used to read the first link register allocated to a node on the Link1(0 + 1). The first zero of “W00001” subsequent to the leading “W” specifies the link1(0 + 1) and the trailing “0001” specifies the address of the link register on the link. (The first link register of the Link2(1 + 1) can be addressed by “W10001”.) In this case, since it is a read access, the link register can be allocated to any of the nodes on the link.

The following example shows how to write a link register.

```
record(longout, “f3rp61_example_32”) {
        field(DTYP, “F3RP61”)
        field(OUT, “@W00001”)
 }
```

The longout record can be used to write the first link register allocated to a node on the Link1(0 + 1). In this case, since it is a write access, the register must be allocated to the F3RP61-based IOC as a node on the link.

Ai / ao, mbbiDirect /mbboDirect are also supported to read / write link registers.

Appendix 1: How to install

1. Building EPICS Base for F3RP61

Un-tar the device support, f3rp61-1.1.1.tgz under appropriate directory, say,

modules/instrument.

Then you'll get the following directory.

modules/instrument/f3rp61-1.1.1

In the directory, you will find the following two configuration files.

CONFIG.Common.linux-f3rp61
CONFIG.linux-x86.linux-f3rp61

Those two files should go under:

base/configure/os.

Move the two files under your base/configure/os first. The two configuration files were created under the following two assumptions.

1. You (will) have installed BSP R2.01 of F3RP61 under /opt of your host computer for cross-development. (/opt is the default installation directory of the installer of the BSP.)
2. Your host architecture is linux-x86.

Modify the two files if necessary.
Next, edit:

base/configure/CONFIG_SITE

in order to add the new cross compiler target architecture, “linux-f3rp61”, to the variable:

CROSS_COMPILER_TARGET_ARCHS.

Now you are ready to build your EPICS base for linux-f3rp61.

2. Building Device / Driver Support Library

Edit a file:

modules/instrument/f3rp61-1.1.1/configure/RELEASE

in order to set EPICS_BASE so that it correctly points to your EPICS base, and then, type “make” under modules/instrument/f3rp61-1.1.1 to build the library.

3. Including Device / Driver Support Library

In your application development directory, edit a file:

(application development top directory)/configure/RELEASE

in order to add a new variable, say, “F3RP61”, which specifies the path to the device / driver support library as follows.

F3RP61 = (where you placed “modules”)/modules/instrument/f3rp61-1.1.1

Next, edit:

(application development top directory)/xxxApp/src/Makefile

in order to add the following four lines at the appropriate positions in the file.

xxx_dbd += f3rp61.dbd

xxx_LIBS += f3rp61

PROD_LDLIBS += – lm3

USR_LDFLAGS += – L(where libm3.so is)

Here, (where libm3.so is) stands for the path to the directory where you (will) have created a soft link to libm3.so.1.0.0 under the name of libm3.so by following the instructions in the Reference Manual of the BSP R2.01 from Yokogawa Electric Corporation. Refer to the following manual for more information on this point.

IM 34M06M51-44E
RTOS-CPU module (F3RP61-□□) Linux BSP Reference Manual
6. PLC Device Access
6.4 User Interface

If you use base Release 3.14.11 or later, the EPICS_HOST_ARCH target build will be executed prior to the build for the cross target, linux-f3rp61, resulting in getting errors. In that case, you can use the conditional directive in the xxxApp/src/Makefile to avoid the problem, as follows.

ifeq ($(T_A),linux-f3rp61)
#Stat of substantial part of the Makefile
#End of substantial part of the Makefile
endif

A sample of Makefile for testApp is included in f3rp61-1.1.1.tgz under the name of:

SampleMakefileForTestApp.

4. Building EPICS Sequencer for F3RP61

Building EPICS Sequencer requires a flex library, libfl.a. The library is missing in the following directory, which was created when you installed the BSP R2.01 (As to BSP R1.01, this was not the case).

/opt/f3rp6x/ppc_6xx/usr/lib

Here, "/opt" is the default installation directory of the installer. If you installed BSP R2.01 under a directory other than there, "/opt" shall be replaced with the directory you chose to install by editing the installer (a script file).

You can find the library, libfl.a, in the BSP. First, visit the following directory.

(BSP)/userland/dev

Here, (BSP) stands for the top directory of un-tared BSP (R2.01). Then, you will find two .tar.gz files under the directory. Next, create a temporary directory and un-tar one of them there. (Whichever is OK, either cf-rootfs-dev-1.tar.gz or nfs-rootfs-dev-1.tar.gz.) The library file, libfl.a, is under

(the temporary directory where you un-tared one of the .tar.gz files)/usr/lib.

From there, copy libfl.a under /opt/f3rp6x/ppc_6xx/usr/lib.

Appendix 2: How to make F3RP61-based IOC real-time

With BSP R2.01 of F3RP61 installed, you can choose to use a kernel with the CONFIG_PREEMPT_RT patch applied. If you choose this option, you might want to choose a priority-based scheduling policy for real-time responsiveness. The choice of the scheduling policy is subject to EPICS base. You can change the value of the variable:

USE_POSIX_THREAD_PRIORITY_SCHEDULING

from the default value of NO to YES in:

base/configure/CONFIG_SITE.

You might also want to call mlockall() in your xxxMain.cpp to make your IOC process memory resident.

Note that you need to be careful so as NOT to run any relevant threads that execute a busy loop if you choose the scheduling policy. Otherwise, what you will have gotten is what you should have gotten.