Device and Driver Support for F3RP61 (Ver. 1.1)

J. Odagiri

Precaution: This version of device / driver support requires BSP Rev2.01 for F3RP61 (e-RT3 2.0 / Linux)

Contents

1. Overview
2. Device Type
3. Accessing I/O Module
4. Handling Special Module
5. Communication with Sequence CPU
6. Cautions in Using F3RP61 and Sequence CPU Side-by-side
7. I/O interrupt Support
8. FL-net Support

Appendix 1: How to install

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

Main changes from the previous version (Rev. 1.0)

- Mode register access support (See, section 3.4)
- New interface for communication with Sequence CPU (See, section 5.1.1)
- 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), made by Yokogawa Electric Corporation. The controller, F3RP61, can access any I/O channel of any I/O module of the FA-M3 PLC on the PLC-bus (base unit). This feature opens way for making an FA-M3 PLC itself a new type of IOC. The device / driver support interfaces the iocCore with the I/O modules as well as ordinary sequence CPUs that works on the same base unit. 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) of F3RP61.

The device / driver support offers only primitive method to access relays and registers of the I/O modules and sequence CPUs. In the sense that any relay and any register 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 the I/O operation, 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.) 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 base unit, the F3RP61 should manage all the I/O activities. If one or more sequence CPUs are on the base unit, some of the I/O modules can be controlled by the sequence CPU and the others can be controlled by 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 6 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 and 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 the I/O modules and the 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) and analog input (ai) records are supported by the device support. Three numbers, unit number, slot number and the 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 mbbiDirect records.

```
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 X1 to X16 are read by 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 value on the input relays is assumed to be

signed one. If the value is unsigned one, “&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 three 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 them, binary output (bo) records, multi-bit binary output direct (mbboDirect) records, long output (longout) records, and analog output (ao) records can be used.

```
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 data bits are written onto Y1 to Y16 by the mbboDirect record.

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 one. If the value is unsigned one, “&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 three 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, serial communication 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 records and mbbiDirect / mbboDirect records are supported. While some of the registers can be 32 bit ones, 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.

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 special character, ‘M’, for the addressing 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_11”) {
        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_12”) {
```

```
        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, which writes a value into the first mode register of the digital input module, 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 writes a value of zero into a hardware register 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 support. At any rate, if you set the conditions on a digital I/O module bit by bit by setting one of the B0, B1, B2, …, BF 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 motion control module in 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 should be able to use interrupt 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 an illustration. The sequencer 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. 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 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 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 of the record for the communication needs to be set to “F3RP61” for the former (synchronous) and “F3RP61Seq” for the latter (asynchronous).

5.1. Communication Based on Shared Memory

The following is the basics to understand how the communication between F3RP61s and sequence CPUs works.

- Each CPU, a sequence CPU or an F3RP61 can have its own region.
- Any CPU, a sequence CPU or an F3RP61 can write only its own region.
- Any CPU, a sequence CPU or an F3RP61 can read any regions.

In order to make the story simple, we consider the case where only one sequence CPU in slot 1 works with only one F3RP61 in slot 2 on the base unit. (See, Fig.1) From the rules mentioned above, how we use the shared memory to make those two CPUs communicate each other is clear.

- If the data go from the sequence CPU (CPU1) to the F3RP61 (CPU2), use the area owned by the sequence CPU (CPU1), and vice versa.

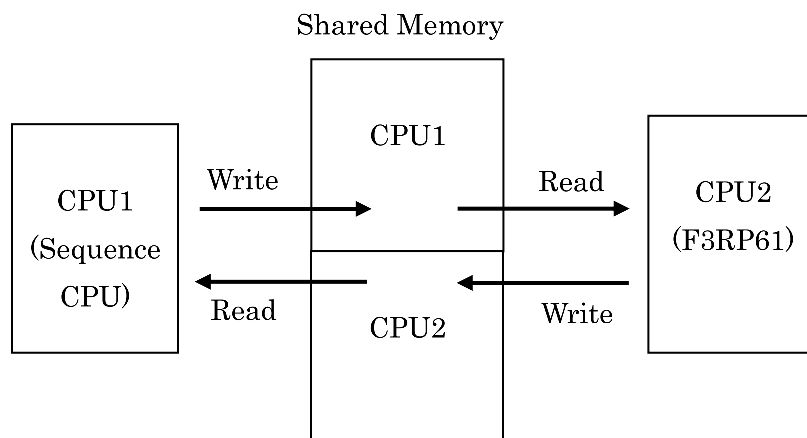

Fig. 1

5.1.1. Communication Based on Shared Memory Using New Interface

From the BSP Rev 2.01 of F3RP61, a set of new interface has been supported to access the shared memory. The device / driver support has also been updated so that they can use the new interfaces. In order to use the new interface, users need to configure the shared memory to specify how many shared relays and shared registers are allocated to each of the CPUs. This allocation can be done by putting the following lines in the startup script for the F3RP61-based IOC. The lines need to be placed before the startup script calls iocInit().

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

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 for CPU1 (0 + 1). The next line means that the same numbers of shared relays and shared registers are allocated for CPU2 (1 + 1).

Note that the users themselves are responsible for making the two configurations, the one done on the F3RP61-side in the startup script and the other done on the sequence CPU-side by using WideField2, to be consistent with each other.

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

```
record(bi, “f3rp61_example_13”) {
        field(DTYP, “F3RP61”)
        field(INP, “@E1”)
}
```

This record reads the first the first shared relay (E1) when the record gets processed. In this case, the relay (E1) can be owned by any CPU as mentioned earlier.

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

```
record(bo, “f3rp61_example_14”) {
        field(DTYP, “F3RP61”)
        field(OUT, “@E1”)
}
```

This record writes the first the first shared relay (E1) when the record gets processed. In

this case, the relay (E1) must be owned by 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_15”) {
        field(DTYP, “F3RP61”)
        field(INP, “@R1”)
 }
```

This record reads the first the first shared register (R1) when the record gets processed. In this case, the register (R1) can be owned by any CPU as mentioned earlier.

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

```
record(longout, “f3rp61_example_16”) {
        field(DTYP, “F3RP61”)
        field(OUT, “@R1”)
 }
```

This record writes the first the first shared register (R1) when the record gets processed. In this case, the register (R1) must be owned by the F3RP61-based IOC as mentioned earlier. Other than longin / longout, ai / ao, mbbiDirect /mbboDirect are supported to read / write shared registers.

5.1.2. Communication Based on Shared Memory Using Old Interface

The author recommends that you chose the new interface for the communication with sequence CPUs since it is much easier to understand. The old interface, however, is still available for backward compatibility. If you use the old interface, the first point you need to know is the following. While the sequence CPU can access the shared memory by bit (shared relay) or by word (shared register), the F3RP61 can access the shared memory only by word.

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 in slot 2 (CPU2). In addition, we assume that we allocate 512 bits (32 words) of shared relays and 256 words of shared registers for both CPU1 (sequence CPU) and CPU2 (F3RP61). In this case, the F3RP61 gets to see the following flat memory space, of which address starts form zero.

```
------------------------------------------------------------------------------------------
R00000: E00001 – E00016          Owned by CPU1 (sequence CPU)
R00001: E00017 – E00032
R00002: E00033 – E00048

             …

R00031: E00497 – E00512
------------------------------------------------------------------------------------------
R00032: E00513 – E00528          Owned by CPU2 (F3RP61)
R00033: E00529 – E00544
R00034: E00545 – E00560

             …

R00063: E00497 – E01024
------------------------------------------------------------------------------------------
R00064: R00001                   Owned by CPU1 (sequence CPU)
R00065: R00002
R00066: R00003

             …

R00319: R00256
------------------------------------------------------------------------------------------
R00320: R00257                   Owned by CPU2 (F3RP61)
R00321: R00258
R00322: R00259

             …

R00575: R00512
------------------------------------------------------------------------------------------
```

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

```
record(mbbiDirect, "f3rp61_example_17") {
        field(DTYP, "F3RP61")
        field(INP, "@CPU1,R0")
}
```

Note that the address in the INP field in this case 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) can access the area only by word when you use the old interface.

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

```
record(mbbiDirect, "f3rp61_example_18") {
        field(DTYP, "F3RP61")
        field(INP, "@CPU1,R65")
}
```

Both R0 and R65 in the last two examples cannot be written by the CPU2 (F3RP61) since they are owned by the sequence CPU1 (sequence CPU).

Next, we consider CPU2 (F3RP61) writing a value into its own region. .

```
record(mbboDirect, "f3rp61_example_19") {
        field(DTYP, "F3RP61")
        field(OUT, "@CPU2,R33")
}
```

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

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

```
record(mbboDirect, "f3rp61_example_20") {
        field(DTYP, "F3RP61")
        field(OUT, "@CPU2,R321")
}
```

Needless to say, both R33 (E00513 – E00528) and R321 (R00257) can be read back by using an appropriate input type record.

You might think that there is no reason to specify the CPU-numbers in the INP fields of the examples shown above since the shared memory is seen by the F3RP61 CPU as a flat space being addressed by a series of sequential address numbers. 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. See the relevant manuals of F3RP61 for more details.

Other record types, ai / ao and longin / longout, are also supported to read / write the shared memory by using the old interface.

5.2. 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 supported by the kernel-level driver. The following example shows how to set (1)/ reset (0) an internal relay ('I') of a sequence CPU in slot 2.

```
record(bo, "f3rp61_example_21") {
        field(DTYP, "F3RP61Seq")
        field(OUT, "@CPU2,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_22") {
        field(DTYP, "F3RP61Seq")
        field(INP, "@CPU2,I4")
}
```

In order to write a data register ('D') of the sequence CPU, the following record can be used.

```
record(longout, "f3rp61_example_23") {
        field(DTYP, "F3RP61Seq")
        field(OUT, "@CPU2,D7")
```

```
}
```

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

```
record(longin, "f3rp61_example_24") {
        field(DTYP, "F3RP61Seq")
        field(INP, "@CPU2,D7")
}
```

Other record types, ai/ao and mbbiDirect/mbboDirect, are also supported to read the data registers. 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.

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

This section gives you cautions in using an F3RP61 CPU and a normal sequence CPU on the same unit. In many cased, this type of multi-CPU configuration is used in case some interlock that is required very high reliability. The sequence CPU is used to handle the interlock I/O signals and dedicates to the work. The F3RP61-based IOC may be used to take care of other control or just to monitor the status of the interlock.

In both cases, the important point is that the sequence CPU must be in the first slot (slot 1) because the CPU in the first slot becomes the master of the unit. The muster CPU resets the whole system upon rebooting.

In the first place, we consider a case where all the I/O modules are subject to the sequence CPU that handles interlock. In this case, an important point is that rebooting the F3RP61 CPU, which is in slot 2 next to the master sequence CPU, can stop the ladder program running on the sequence CPU if the F3RP61 has had a direct access to

the I/O module for the reason described below.

When 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 recognize the F3RP61 as one of their masters reset themselves when they detect the signal. This makes the I/O modules un-accessible by the sequence CPU and makes the ladder program stop with an I/O error.

For this reason, it is recommended that you make the F3RP61 CPU read the relay status indirectly via some internal devices ('I', 'D', 'B') of the sequence CPU or the shared memory ('E', 'R') by using one of the methods described in the previous section.

Next, we consider a case where some of the I/O modules are subject to the sequence CPU and the others are subject to the F3RP61 CPU. The point here is that you need to tell the sequence CPU not to touch the I/O modules which are subject to the F3RP61 CPU. This setting can be done on the sequence CPU by using WideField2.

From the main menu, "Project", select "Configuration" and then, "DIO Setting". Change the default setting from "Used" to "Unused" for the I/O modules which subject to the F3RP61 CPU. Otherwise, the sequence CPU overwrites the I/O channels with the value of zero even though I/O execution commands for the I/O modules do not appear explicitly in the ladder program.

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 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.

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

```
record(ai, "f3rp61_example_25") {
        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, you can write output data into the first data register (A1) of the D/A module upon the same interrupt by using the following record.

```
record(ao, "f3rp61_example_26") {
        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 F3RP61 can respond to interrupts.

```
record(bi, "f3rp61_example_27") {
        field(DTYP, "F3RP61")
        field(SCAN, "I/O Intr")
        field(INP, "@U0,S2,X1:U0,S2,X1")
}
```

This bi record reads the status of the input relay onto which the trigger signal goes. 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 "on" (1). Otherwise, it reads "off" (0). By changing the pulse duration with checking the record value, you can see how long it takes for the record to get processed starting at the interrupt.

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. For details about FL-net, please consult relevant manuals from Yokogawa Electric Corporation.

How to use FL-net on F3RP61-based IOC with cyclic transmission is very similar with how to use shared memory with the new interface. Users need to configure the link memory to specify how many link relays and link registers are allocated to each of the CPUs. This allocation can be done by putting the following lines in the startup script for the F3RP61-based IOC. The lines need to be placed before the startup script calls iocInit().

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

The first line implies that 512 bits of link relays and 256 words of link registers are allocated for Link1 (0 + 1). The next line means that the same numbers of link relays and link registers are allocated for Link2 (1 + 1).

Note that the users themselves are responsible for making the two configurations, the one done on the F3RP61-based IOC with the startup script and the other done on the sequence CPU with the ladder program development tool, WideField2, to be consistent with each other.

The following example shows how to read a link relay.

```
record(bi, “f3rp61_example_28”) {
          field(DTYP, “F3RP61”)
          field(INP, “@L1”)
 }
```

This record reads the first the first link relay (L1) when the record gets processed. In this case, the relay (L1) can be owned by any CPU as mentioned earlier.

The following example shows how to write a link relay.

```
record(bo, “f3rp61_example_29”) {
```

```
        field(DTYP, "F3RP61")
        field(OUT, "@L1")
}
```

This record writes the first the first link relay (L1) when the record gets processed. In this case, the relay (L1) must be owned by the F3RP61 CPU as mentioned earlier.

The following example shows how to read a link register.

```
record(longin, "f3rp61_example_30") {
        field(DTYP, "F3RP61")
        field(INP, "@W1")
}
```

This record reads the first the first link register (W1) when the record gets processed. In this case, the register (W1) can be owned by any CPU as mentioned earlier.

The following example shows how to write a link register.

```
record(longout, "f3rp61_example_31") {
        field(DTYP, "F3RP61")
        field(OUT, "@W1")
}
```

This record writes the first the first link register (W1) when the record gets processed. In this case, the register (W1) must be owned by the F3RP61 CPU as mentioned earlier. Ai / ao, mbbiDirect /mbboDirect are also supported to read / write link registers.

Appendix 1: How to install

In the following explanation, it is assumed that relevant files under base/configure have already been added or modified to build the base for the target, F3RP61. A sample of those files is available on the page from which you downloaded this manual. Please

check it first.

The device / driver support is supposed to be placed under:

modules/instrument.

Make a new directory, say, f3rp61 and then move to the directory.

modules/instrument/f3rp61

Copy f3rp61-1.1.0.tgz under the directory and decompress it there. Then, you'll have the following directory.

modules/instrument/f3rp61/f3rp61-1.1.0

In the above directory, where you'll see Make file, type "make - target_arch" to build the library. Here, "target_arch" is the architecture name set to the variable, CROSS_COMPILER_TARGET_ARCHS in base/configure/CONFIG_SITE. All the other procedure is the same as in other EPICS libraries.

In order to use the library in your application development directory, in the first place, you need to edit a configuration file, configure/RELEASE, to specify the path to reach the device / driver support library by setting a variable, say, F3RP61.

F3RP61 = "where you placed modules"/modules/instrument/f3rp61/f3rp61-1.1.0

In addition, in xxxApp/src/Makefile, you need to add the following two lines:

xxx_dbd += f3rp61.dbd
xxx_LIBS += f3rp61

The following two lines are also needed in order to link the library, libm3.so, which is included in the BSP of F3RP61.

PROD_LDLIBS += – lm3
USR_LDFLAGS += – L"where libm3.so is"

Here, "where libm3.so is" stands for the directory where you have created the soft link to libm3.so.1.0.0 included in the BSP. Please refer relevant manuals from Yokogawa

Electric Corporation for more details on the library, libm3.so.

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

From the BSP Rev 2.01 of F3RP61, you can choose to use a kernel with CONFIG_PREEMPT_RT 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 not to the device / driver support but to EPICS base. You can change a variable, USE_POSIX_THREAD_PRIORITY_SCHEDULING, from the default value of NO to YES in base/configure/CONFIG_SITE. That is all to make your F3RP61-based IOC real-time.

Note that you need to be careful so as not to run a thread that executes a busy loop if you choose the scheduling policy. Otherwise, what you will have gotten is what you should have gotten.