The Hardware

Two kind of protecting circuits, both based on commercial elements, were implemented in the module. The first one uses the LT4356 Surge Stopper from the Linear Technology. It detects over-voltage and over-current conditions and it is capable to sustain an acceptable voltage level for defined time interval, after which the channel is switched off; in this way current transients can be ignored. The NMOS transistor with very low value of the Ron is used as a switch. The LT4356 protection is installed in seven channels where expected ranges of supply current changes are large, i.e. in the Digital and Analog domains. Remained channels (Gate and Steering Domains) exploit a voltage monitor LTC2912 from Linear Technology, driving a Solid State Relay (SSR). After detecting an error condition the given channel (and a whole domain) are switched off.

The over-voltage checks

The checks performed in the module are listed in the following table:

CH	NET (...)	CHECKED	CHECK DESCRIPTION	REL	LIMIT_L	LIMIT_H	HARDWARE
DV1	DHP_IO	itself	DHP_IO	<		2.3	LT4356
DV2	SW_DVDD	itself	SW_DVDD	<		2.3	LT4356
DV3	DHP_CORE	itself	DHP_CORE	<		1.7	LT4356
DV4	DCD_DVDD	itself	DCD_DVDD	<		2.3	LT4356
AV1	REF_IN	itself	REF_IN	<		1.6	LT4356
AV2	VSOURCE	itself	VSOURCE	<		7.5	LT4356
AV3	AMP_LOW	itself	AMP_LOW	<		2	LTC2912
AV4	DCD_AVDD	itself	DCD_AVDD	<		2.3	LT4356
ST1	Clear_ON	itself	Clear_ON	<		26	LTC2912
ST2	SW_Refln	ST2,ST4	SW_Refin-SW_SUB	<		2.3	LTC2912
ST3	Clear_OFF	itself	Clear_OFF	<,<	-1	6	LTC2912
ST4	SW_SUB	ST9,ST4	Gate_ON_1-SW_SUB	>	-0.6		LTC2912
ST5	free	ST11,ST4	Gate_ON_3-SW_SUB	>	-0.6		LTC2912
ST6	HV	ST10,ST4	Gate_ON_2-SW_SUB	>	-0.6		LTC2912
ST7	VBulk	itself	Vbulk	<		17.5	LTC2912
ST8	VGuard	itself	Vguard	<,<	-9	1	LTC2912
ST9	Gate_ON_1	itself	Gate_ON_1	>	-14		LTC2912
ST10	Gate_ON_2	itself	Gate_ON_2	>	-14		LTC2912
ST11	Gate_ON_3	itself	Gate_ON_3	>	-14		LTC2912
ST12	Gate_OFF	itself	Gate_OFF	<.<	-4.5	6	LTC2912
ST13	CCG1	itself	CCG1	<,<	-12	2.3	LTC2912
ST14	CCG2	itself	CCG2	<,<	-12	2.3	LTC2912
ST15	CCG3	itself	CCG3	<,<	-12	2.3	LTC2912
ST16	VDrift	itself	Vdrift	>	-14.3		LTC2912

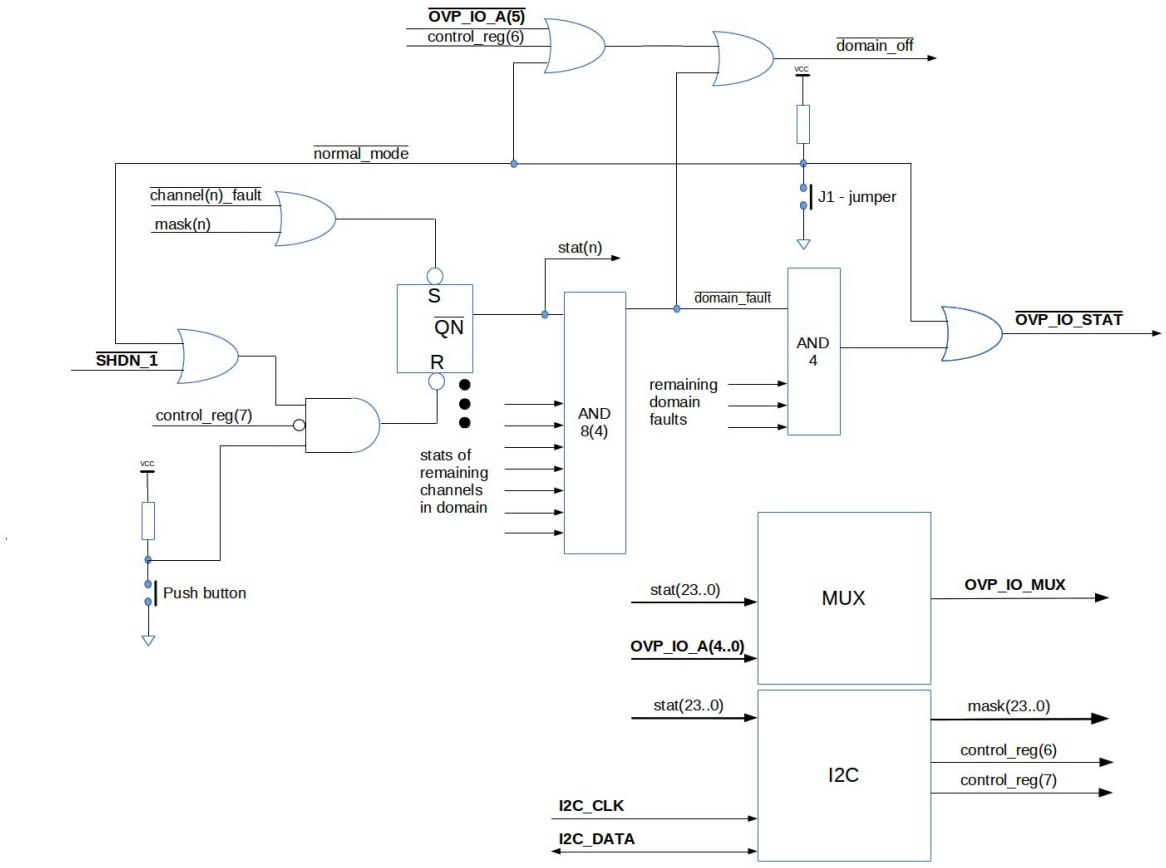
< : voltage less then LIMIT_H;

<,< : voltage in wrange from LIMIT_L to LIMIT_H;

> : voltage greater than LIMIT_L.

The control logic

The control logic of the module has been implemented in two XC95144 CPLD components from Xilinx. One of them performs all logical functions relevant to faults detection. The other contains the I2C interface. Both chips are connected with custom serial link.



Drawing 1: Block diagram of the control logic.

The Status Register

The Status Register has 24 bits and each of them corresponds to single supply channel:

Bits 23:16	Bits 15:8	Bits 7:4	Bits 3:0
GT domain checks	ST domain checks	ANALOG domain checks	DIGITAL domain checks
ST[16:9]	ST[8:1]	AV[3:0]	DV[3:0]

Over-voltage fault effects in the corresponding bit zeroed. Then the whole domain to which the faulty channel belongs is switched off. Logical OR of all 24 bits of the Status Register sources the General Fault signal, which is sent to the external pin OVP_IO_STAT.

Individual bits of the Status Register can be read out in two ways:

- from the OVP_IO_MUX external pin. There the value of Status Register bit pointed by the OVP_IO_A(4:0) external address pins is present;
- using the I2C read operation.

Recovering from the error state of the module must be performed in two steps:

- setting all channel voltages to acceptable values;
- resetting the module.

There are three possible sources of the reset necessary for recovering:

- push-button on the board;
- external pin SHDN_1 (active LOW), only when the jumper J1 is ON (normal mode operation);
- I2C Control register bit 7, active HIGH.

Debugging

For every bit of the Status register there exist a corresponding bit of the so called Mask register. When set to HIGH, the given Status bit does not participate in the over-voltage detection process. See the Drawing 1.

A few other provisions have been also made to simplify the debugging of the module:

J1 removed ("engineering mode")	General Fault ($\overline{\text{OVP\_IO\_STAT}}$) disabled Voltages are NOT blocked.
Bit 6 of the I2C Control register set HIGH	Voltages are NOT blocked.
OVP_IO_A(5) bit set LOW	Voltages are NOT blocked.

Note, that for the normal operation the jumper J1 should be always plugged in and the OVP_IO_A(5) pin set to HIGH.

I2C Interface

The I2C interface has been used with the USB Interface Adapter Evaluation Module from Texas Instruments, based on the TI TUSB3210 USB peripheral chip.

The I2C should operate at default 100kHz speed.
The I2C address of the module is fixed as **0x3C**.

The I2C interface allows access to the Status, Mask and Control registers; the latter one is 8-bit long:

RESET	BLOCK	0	0	0	0	0	LIVE

Bit 7 (RESET) is used for resetting the Status Register (operation equivalent to asserting the SHDN_1 or pressing the Push Button).
Bit 6 (BLOCK) is used to keep the module in the "blocking mode", in which the output voltages are not disconnected after the fault is detected.
Bit 0 (LIVE) , when set, allows to read directly the levels of the channels fault signals instead of their registered values.
Both bits: BLOCK and LIVE are foreseen only for debugging and should never be set in normal operation.

I2C read

The read operation should be always performed as **an eight byte long I2C transfer from offset 0**.

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0xFF or 0x00	stat[23:16]	stat[15:8]	stat[7:0]	cont[7:0]	mask[23:16]	mask[15:8]	mask[7:0]

Byte 0 contains the general fault bit repeated eight times (in case of error it results in value of 0xFF, otherwise it is equal 0).
Bytes 1:3 show actual value of the Status Register. Here registered faults can be identified.
Byte 4 contains actual value of the I2C Control Register.
Bytes 5:7 contain value of the Mask register.

I2C WRITE

The write operation should be always performed as **a four byte long I2C transfer with register offset equal 4**. Bytes content is shown in the following table:

Byte 0	Byte 1	Byte 2	Byte 3
cont[7:0]	mask[7:0]	mask[15:8]	mask[23:16]

Appendix A

Extract from the C# control program operating with the TI I2C Interface. The *usa.i2cWrite* and the *usa.i2cRead* calls came from the relevant library provided with the device.

Write operation:

```
private void button1_Click(object sender, EventArgs e)
    {
        Int32 i, wdata;
        byte[] data = { 0x0A, 0x0B, 0x0C, 0x0D };
        byte woff, cbyte;
        CheckBox cb;
        UInt32 maskdata = 0, msk, cntdata = 0;
```

```
        byte[] mdata = new byte[4];

        //prepare mask
        for (i = 0, msk = 1; i < 24; i++)
        {
           cb = (CheckBox)MASKR.Controls[i]; //
           maskdata |= (cb.Checked) ? msk : 0;
           msk <<= 1;
        }

        mdata[0] = (byte)(maskdata & 0xFF); //
        mdata[1] = (byte)((maskdata & 0xFF00) >> 8); //
        mdata[2] = (byte)((maskdata & 0xFF0000) >> 16); //
        mdata[3] = 0;

        //mask ready
        //now control
        for (i = 0, msk = 1; i < 8; i++)
        {
           cb = (CheckBox)CONTROL.Controls[i];
           cntdata |= (cb.Checked) ? msk : 0;
           msk <<= 1;
        }

        cbyte = (byte)(cntdata & 0xFF);
       //control ready

           data[0] = cbyte; //control register, 0x80 corresponds to RESET bit ON
           data[1] = mdata[2];//AN, DV, bit 0 corresponds to AV4 (next come: AV3,AV2,AV1,DV4,DV3,DV2,DV1)
           data[2] = mdata[1];//ST8-ST1, bit 0 to ST8
           data[3] = mdata[0];//ST16-ST9, bit 0 to ST16

           woff=4;

           if (usa.i2cWrite(0x3C, woff, 4, data) == 0) statBox.Text = "WRITE OK";
           else
              statBox.Text = "WRITE ERROR";
        }

     }
```

Read operation:

```
 private void button2_Click(object sender, EventArgs e)
     {
        Int32 i;
        byte[] rdata = new byte[8];
        byte[] statdata = new byte[4];
        byte[] maskdata = new byte[4];

        byte roff;
        UInt32 status, mask;
        CheckBox cb;

         roff=0;

           if (usa.i2cRead(0x3C, roff, 8, ref rdata) == 0) statBox.Text = "READ OK";   //eight bytes from offset 0
           else
                                                    statBox.Text = "READ ERROR";

           statdata[0] = rdata[3];
           statdata[1] = rdata[2];
           statdata[2] = rdata[1];
           statdata[3] = 0;
```

```
        status = 0xFFFFFF & BitConverter.ToUInt32(statdata, 0);

        statBox.Text = System.String.Format("{0,6:X6}", status);

        for (i = 0; i < 24; i++)
        {
           cb = (CheckBox)STATUS.Controls[i];
           cb.Checked = ((status & (1 << i)) == 0) ? false : true;
        } //here the status check boxes are filled: bit 0: DV1, bit 1: DV2, bit 2: DV3, bit 3: DV4, bit 4: AV1 and so on

        maskdata[0] = rdata[7];
        maskdata[1] = rdata[6];
        maskdata[2] = rdata[5];
        maskdata[3] = 0;

        mask = 0xFFFFFF & BitConverter.ToUInt32(maskdata, 0);
        for (i = 0; i < 24; i++)
        {
           cb = (CheckBox)MASKR.Controls[i];
           cb.Checked = ((mask & (1 << i)) == 0) ? false : true;
        }

     }
  }
```