This thread has been locked.
If you have a related question, please click the "Ask a related question" button in the top right corner. The newly created question will be automatically linked to this question.

CC1101: stuck waiting for CC1101 to bring GDO0 low (with IOCFG0=0x06), why?

Resolved

Britton Kerin

Part Number: CC1101

I'm using the defualt Smart-RF supplied settings, including IOCFG0 = 0x06. For this setting the datasheet says in table 41:

> Asserts when sync word has been sent / received, and de-asserts at the end of the packet. In RX, the pin will also de-assert when a packet is discarded due to address or maximum length filtering or when the radio enters RXFIFO_OVERFLOW state. In TX the pin will de-assert if the TX FIFO underflows.

Nevertheless, my code blocks forever waiting for GDO0 to go low in it's receive_packet routine. It does this right about the time that overflow will occur (I watch RXBYTES increase packet-to-packet and this problem manifests when FIFO is almost full). If I keep the packet send rate low enough that overflow doesn't happen this problem doesn't show up.

The above datasheet section seems to be saying that GDO0 should never stay high forever: it should go low due to overflow in this situation, and after this I expect it should not go high again while in overflow state regardless of received sync word or other events until SFRX. Is this not the case? How does it end up stuck high?

Thanks

Britton Kerin

Andrew G

Don't know the exact answer to your question, but for CC1200 I use

IOCFG0 = 0x01 // RXFIFO_THR_PACKET
FIFO_CFG = 0xff // CRC flush, high RX threshold

and that seems to work OK for receiving packets.

Andrew G

Also, are you sure you are blocked on GDO0 going low and not high? Perhaps the receiver enters a state where it doesn't receive sync words anymore?

Britton Kerin

In reply to Andrew G:

I'm sure it's blocked waiting for GDO0 to go low, and it's polling for low and not edge-triggered. The problem seems more likely to be the opposite: after the overflow sends it low and puts the chip in state RXFIFO_OVERFLOW, it subsequently manages to receive another sync word and bring GDO0 high again, then sticks there.

I have some more careful instrumentation and scope work to verify this but it's the best guess I've got so far.

Britton Kerin

I'm wondering if my CC1101s might be broken clones. Please TI can you comment on the below behavior and whether it is something genuine TI chips exhibit?

The CC1101 definitely doesn't seem to be behaving as advertised on RXFIFO overflow.

As I understand, at least the following things are supposed to happen on Rx overflow:

* GDO0 goes low (when IOCFG0=0x06)
* MARCSTATE goes to 0x11 (RXFIFO_OVERFLOW state)
* RXBYTES goes to something with bit 7 set (probably 11000000)

What actually seems to happen:

* GDO0 does not go low

* MARCSTATE stays 0x0d (RX state)
* RXBYTES stays at 0x41 (decimal 65, binary 01000001)

I guess the RXFIFO_OVERFLOW state can only be exited by issuing SFRX strobe, The datasheet doesn't say this explicitly as it does for TXFIFO_UNDERFLOW and SFTX strobe, but the state diagrams imply it and it seems like a reasonable assumption. However, if RXFIFO_OVERFLOW can be exited other ways I guess there is some chance this is what is going on?

This behavior seems easy to reproduce. I just send packets somewhat faster than they are received, causing accumulation in RXFIFO until overflow hits and the behavior is triggered. I'm using the default SmartRF-produced register settings for 433 MHz carrier, 26MHz crystal, 1.2kBaud GFSK optimized for current, but with transmit power setting PATABLE[0] = 0x34 (~ -10 dBm) to avoid near-in saturation.

Here is a log showing the output of my test receiver program:

```
Configuration Register Values
-----------------------------
IOCFG2 : 0x29
IOCFG1 : 0x2e
IOCFG0 : 0x06
FIFOTHR : 0x47
SYNC1 : 0xd3
SYNC0 : 0x91
PKTLEN : 0xff
PKTCTRL1 : 0x04
PKTCTRL0 : 0x05
ADDR : 0x00
CHANNR : 0x00
FSCTRL1 : 0x06
FSCTRL0 : 0x00
FREQ2 : 0x10
FREQ1 : 0xa7
FREQ0 : 0x62
MDMCFG4 : 0xf5
MDMCFG3 : 0x83
MDMCFG2 : 0x93
MDMCFG1 : 0x22
MDMCFG0 : 0xf8
DEVIATN : 0x15
MCSM2 : 0x07
MCSM1 : 0x30
MCSM0 : 0x18
FOCCFG : 0x16
BSCFG : 0x6c
AGCCTRL2 : 0x03
AGCCTRL1 : 0x40
AGCCTRL0 : 0x91
WOREVT1 : 0x87
WOREVT0 : 0x6b
WORCTRL : 0xfb
FREND1 : 0x56
FREND0 : 0x10
FSCAL3 : 0xe9
FSCAL2 : 0x2a
FSCAL1 : 0x00
FSCAL0 : 0x1f
RCCTRL1 : 0x41
RCCTRL0 : 0x00
```

```
FSTEST : 0x59
PTEST : 0x7f
AGCTEST : 0x3f
TEST2 : 0x81
TEST1 : 0x35
TEST0 : 0x09

Status Register Values
----------------------
PARTNUM : 0x00
VERSION : 0x14
FREQEST : 0x00
LQI : 0x56
RSSI : 0x80
MARCSTATE : 0x01
WORTIME1 : 0x00
WORTIME0 : 0x00
PKTSTATUS : 0x00
VCO_VC_DAC : 0x94
TXBYTES : 0x00
RXBYTES : 0x00
RCCTRL1_STATUS : 0x00
RCCTRL0_STATUS : 0x00

carrier freq fields: 0x10 0xa7 0x62
cfreq: 432204696
Setting Tx power to approximately -10 dBm
Calibration done, in theory

About to start listening

wait for !GDO0
got !GDO0
Got packet
length: 3
data[0]: 42
data[1]: 111
data[2]: 110
pn: 28526
crc_ok: 128
rssi: 86
lqi: 31

wait for !GDO0
got !GDO0
Got packet
length: 3
data[0]: 42
data[1]: 111
data[2]: 111
pn: 28527
crc_ok: 128
rssi: 87
lqi: 30

wait for !GDO0
got !GDO0
Got packet
length: 3
data[0]: 42
data[1]: 111
```

```
data[2]: 112
pn: 28528
crc_ok: 128
rssi: 86
lqi: 31

wait for !GDO0
got !GDO0
Got packet
length: 3
data[0]: 42
data[1]: 111
data[2]: 113
pn: 28529
crc_ok: 128
rssi: 86
lqi: 35

wait for !GDO0
got !GDO0
Got packet
length: 3
data[0]: 42
data[1]: 111
data[2]: 114
pn: 28530
crc_ok: 128
rssi: 87
lqi: 34

wait for !GDO0
got !GDO0
Got packet
length: 3
data[0]: 42
data[1]: 111
data[2]: 115
pn: 28531
crc_ok: 128
rssi: 86
lqi: 34

wait for !GDO0
got !GDO0
Got packet
length: 3
data[0]: 42
data[1]: 111
data[2]: 116
pn: 28532
crc_ok: 128
rssi: 87
lqi: 32

wait for !GDO0
got !GDO0
Got packet
length: 3
data[0]: 42
data[1]: 111
data[2]: 117
```

```
pn: 28533
crc_ok: 128
rssi: 87
lqi: 32

wait for !GDO0
got !GDO0
Got packet
length: 3
data[0]: 42
data[1]: 111
data[2]: 118
pn: 28534
crc_ok: 128
rssi: 87
lqi: 30

wait for !GDO0
got !GDO0
bytes ready to read > 58
Got packet
length: 3
data[0]: 42
data[1]: 111
data[2]: 119
pn: 28535
crc_ok: 128
rssi: 86
lqi: 32

wait for !GDO0

>100042 waits
MARCSTATE: 0x0d
RXBYTES: 0x41

>100042 waits
MARCSTATE: 0x0d
RXBYTES: 0x41

>100042 waits
MARCSTATE: 0x0d
RXBYTES: 0x41

[repeats forever]
```

Andrew G

In reply to Britton Kerin:

Could it be the sender that overflows first? If so, you wouldn't see any packets on the RX side until the sender is flushed.

Do you want to post your code for the send and receive functions? It's hard to tell what's going on without seeing the code.

Britton Kerin

In reply to Andrew G:

Good thought but the sender is not reporting any problem:

My send_packet() starts with:

```
assert (cc1101_get_state () == CC1101_STATE_TX);
```

and ends with:

```
// If the sender behaves it should never cause underflow, or leave bytes
// in the FIFO.
#ifndef NDEBUG
uint8_t txbytes = CC1101_READ_STATUS_REG_SAFELY (CC1101_TXBYTES);
#endif
assert (! (txbytes & CC1101_TXBYTES_TXFIFO_UNDERFLOW_MASK));
assert ((txbytes & CC1101_TXBYTES_NUM_TXBYTES_MASK) == 0);
```

I get to overflow by pausing the Rx microcontroller in Rx mode before reading the FIFO for longer than I pause at the Tx between packets, hence the gradual approach to overflow.

There's some context for my send and receive that make them not fully representative
of the situation by themselves. I'll try to boil it all down to a simpler test case soon.
However, the code that produces the invalid MARCSTATE/RXBYTES combination in the
above output is quite simple:

```
printf (">100042 waits\n"); \
uint8_t XxX_ms = CC1101_READ_STATUS_REG_SAFELY (CC1101_MARCSTATE); \
printf ("MARCSTATE: 0x%02" PRIx8 "\n", XxX_ms); \
uint8_t XxX_rxbytes = CC1101_READ_STATUS_REG_SAFELY (CC1101_RXBYTES); \
printf ("RXBYTES: 0x%02" PRIx8 "\n", XxX_rxbytes);
```

Which gives output:

```
>100042 waits
MARCSTATE: 0x0d
RXBYTES: 0x41
```

So far as I understand this situation is completely invalid and should never occur. The low 7 bits of RXBYTES should never read > 64 (> 0x40), and if because of overflow they do, then MARCSTATE should certainly be RXFIFO_OVERFLOW, not RX (0x11, not 0x0d). Note that the above output repeats forever once the CC1101 gets stuck, so it cannot be a race between the query of MARCSTATE and that of RXBYTES.

CC1101_READ_STATUS_REG_SAFELY() just implements the read-until-get-same-value-twice-in-a-row strategy to guard against corrupt reads as described in the silicon errata for the CC1101. It works as expected elsewhere, but here it is in expanded terms (the remaining undefined funtions in cc1101_read_reg() are heavily used in this program and elsewhere and seem to work fine everywhere):

```
#define CC1101_TRANSFER_TYPE_READ_BURST 0xC0

#define CC1101_REGISTER_TYPE_STATUS CC1101_TRANSFER_TYPE_READ_BURST

// Read CC1101 status register
#define CC1101_READ_STATUS_REG_SAFELY(address) \
cc1101_read_reg_safely (address, CC1101_REGISTER_TYPE_STATUS)

uint8_t
cc1101_read_reg (uint8_t address, uint8_t type)
{
uint8_t typed_address = address | type; // Because CC1101 works this way
CC1101_SELECT ();
WAIT_UNTIL_MISO_LOW ();
spi_transfer (typed_address); // Send typed address
uint8_t const unused_value = 0x00;
uint8_t val = spi_transfer (unused_value);
CC1101_DESELECT ();

return val;
}

static uint8_t last_config_reg = 0x2E;

uint8_t
cc1101_read_reg_safely (uint8_t address, uint8_t type)
{
// The synchronization bug we're addressing doesn't affect static
// configuration registers (registers in the 0x00 to 0x2E range).
// FIXME: so just require an address in the affected range and rename the
// function, since this issue isn't hidden from the interface anyway.
if ( address <= last_config_reg ) {
assert (type == CC1101_REGISTER_TYPE_CONFIG);
return cc1101_read_reg (address, type);
}

// Repeatedly read the register until we get the same value twice in a row
uint8_t
orv = cc1101_read_reg (address, CC1101_REGISTER_TYPE_STATUS),
nrv = cc1101_read_reg (address, CC1101_REGISTER_TYPE_STATUS);
while ( nrv != orv ) {
orv = nrv;
nrv = cc1101_read_reg (address, CC1101_REGISTER_TYPE_STATUS);
}

return nrv;
}
```

Britton Kerin

Well looks like I just rediscovered the "RXFIFO_OVERFLOW Issue" that is clearly documented in the CC1101 silicon errata. Sorry folks, hard to keep all the errata in mind at once I guess.

Turning off PKTCTRL1.APPEND_STATUS gives the expected rx fifo overflow behavior for me (WARNING: but there are other ways to get the same sort of problem as documented in the silicon errata).

This thread has been locked.
If you have a related question, please click the "Ask a related question" button in the top right corner. The newly created question will be automatically linked to this question.