FCDEV3B User's Manual

Version 1.28, last edited 2019/04/15

1. Hardware setup

1.1. Power supply connection

The GSM modem chipset on the FCDEV3B is designed to run on battery power, typically a single Li-Ion cell which produces a voltage of 4.2 V when fully charged, lowering as it discharges. However, the chipset uses only LDO regulators and no switchers, hence when a fixed-output power supply is used instead of an actual battery, running at a lower voltage will result in less heat dissipation and thus greater reliability and longevity. The standard recommended voltage for fixed supply operation is 3.6 V.

The power input connector on the FCDEV3B is a Weidmuller 1510460000 (reference designator J305, the big orange connector readily stands out visually); there are several mating connector parts available from Weidmuller; Phoenix Contact also makes compatible connectors. Unless you have bought a ready-made power supply with your board, you will need to make your own power connection cable. If you are connecting a ready-made cable, the connector only goes in one way, but if you are wiring your own cable, you need to observe the polarity: the positive needs to be on pin 1 (facing inward), the negative needs to be on pin 3 (facing toward the edge of the board), and the middle pin is unused.

If you are connecting an actual battery rather than a power supply, presumably to test truly mobile operation with cell reselection, location updates and handovers, please note that the FCDEV3B will only draw power from that battery and has no provision for charging. Therefore, you will need to use an external charger and connect a fully charged battery for the duration of your mobility tests.

1.2. Serial port connection

The primary means of controlling, programming and communication with the FCDEV3B is a pair of low-voltage serial interfaces, also known as UARTs. Each of the two UARTs is natively a low-voltage interface (2.8 V native logic levels, tolerant of 3.3 V, but NOT any higher voltages) and can be converted externally to RS-232 or to USB as needed for each user's application.

With our standard firmware UART 0 (called MODEM in the Calypso chip documentation) provides a standard ASCII AT command interface (see Chapter 2), and UART 1 (called IrDA in the Calypso chip docs, although we are not using it for that purpose) provides a debug trace and command interface in TI's binary packet format (see Chapter 3). If you are going to be running your own firmware instead of ours, you can naturally use each UART however you like, but in any case you will need to have at least one UART connected in order to make the modem board do anything useful.

The dual UART header on the FCDEV3B is J301, and it has the following pinout (pin 1 is toward the power input connector):

GND	1	2	GND
TX_IRDA	3	4	TX_MODEM
RX_IRDA	5	6	RX_MODEM
NC	7	8	RTS_MODEM
NC	9	10	CTS_MODEM

If you are going to connect to both Calypso UARTs by way of an FT2232D USB dual UART, the convention established by the Mother of FreeCalypso is that Calypso's MODEM UART should be connected to

FT2232D channel A, and the IrDA UART should be connected to channel B. Connect jumper wires as follows:

FT2232D signal	Calypso signal
ADBUS0	RX_MODEM
ADBUS1	TX_MODEM
ADBUS2	CTS_MODEM
ADBUS3	RTS_MODEM
BDBUS0	RX_IRDA
BDBUS1	TX_IRDA

Be sure to connect the ground as well! The FT2232D adapter boards sold by FreeCalypso have two ground pins, one on the ADBUS side and the other on the BDBUS side, which naturally map to the two ground pins on the J301 dual UART header on the FCDEV3B.

If you use an FT2232D adapter as described above to connect the FCDEV3B's UARTs to a Linux PC or laptop, you will have a single USB device with two *ttyUSBx* devices behind it; if you have no other *ttyUSBx* devices, the Calypso's MODEM UART (the one we use for the standard ASCII AT command interface) will appear to your Linux host as `/dev/ttyUSB0`, and the Calypso's IrDA UART (the one we use for the debug trace and command binary packet interface) will appear as `/dev/ttyUSB1`.

1.3. Voice interfaces

The UARTs described in the previous section are sufficient for all data, SMS and ''hacking'' applications except voice. While the AT command interface provided by our standard firmware and accessible via those UARTs can dial and answer GSM voice calls, actually hearing or otherwise receiving the voice call downlink and sending your own voice into the call uplink requires additional interfaces. There are two voice interface options available on the FCDEV3B:

- The standard analog earpiece and microphone interfaces provided by the Calypso+Iota chipset have on-board loudspeaker driver and microphone input circuits connected to them on our FCDEV3B. You can connect a loudspeaker to two-post header J312 and an electret condenser microphone to two-post header J313, and then use those acoustic transducers to exercise GSM voice calls. See Appendix A for more information.
- The Calypso chip has a poorly documented provision for a digital voice PCM interface, and this interface is brought out to a header on the FCDEV3B. Despite the lack of documentation from TI, we have now successfully reverse-engineered this Calypso digital voice PCM interface and implemented proper support for it in our firmware — see Appendix B for more information.

1.4. Antenna connection

The gold-plated SMA connector J308 that hangs over the edge of the board is where you need to connect the GSM antenna; the modem will not receive any GSM signals without an antenna connected. Alternatively one can connect a coaxial cable going to a base station simulator.

1.5. SIM socket

The SIM socket on the board is a C&K CCM03-3013 (reference designator J302); it accepts the 2FF classic SIM size that culturally corresponds to traditional GSM/2G.

1.6. Power on/off control

Having battery voltage present at power input connector J305 will not in itself cause the Calypso chipset to power on and boot up — it also needs to sense a low on the PWON control input. The FCDEV3B provides two ways to control the power-on sequence:

- When battery power is applied to the board but the green LED is off, pressing the PWON button will cause the chipset to power on and boot up, and the green LED will turn on.

- If the board is to be used in an unattended environment where requiring a human to press a button is not acceptable, one can put a shorting block on two-post jumper header JP1. This two-post header is wired in parallel with the PWON button, hence putting a shorting block on it is equivalent to having the button permanently held down. If the PWON button is permanently held down or JP1 is shorted, the chipset will power on and boot immediately as soon as battery power is applied.

The green LED on the board indicates the on or off state of the chipset: this LED lights up when the power-on control in the Iota chip is in the switched-on state, and is off otherwise. Once the chipset has powered on and the green LED is lit, a return to the switched-off state can be commanded only through software, and if JP1 is shorted, a software power-off command turns into a reset operation as the switch-off will be immediately followed by another switch-on.

There is also a RESET button on the board that shorts the nTESTRESET signal to ground; pressing this button will cause the chipset to switch on and reset irrespective of its previous state.

1.6.1. Sleep mode considerations

If you are going to operate the board with JP1 shorted for automatic unattended switch-on, you need to be aware of sleep mode implications. Our current V2 boards have working Calypso sleep modes, and our current firmware versions make use of all of them. In the most complete power saving mode known as deep sleep, which is normally entered between paging wake-up times when the modem is camped on a cell but is otherwise idle, the main 26 MHz VCXO that provides all system clocks is stopped, leaving only the 32.768 kHz clock provided by a separate crystal oscillator in the RTC power domain. During these deep sleep times the system power consumption is reduced to the absolute minimum, and normally this deep sleep mode is accompanied by the LDO regulators in the Iota ABB being switched into their own sleep modes as well, to avoid power waste within the regulators themselves. However, the latter ABB sleep mode is not possible when the PWON control input is held low (it is a wake-up condition for the ABB), hence if JP1 is shorted, our firmware won't put the Iota ABB into its sleep mode: it will still put the Calypso into deep sleep and stop the VCXO, but no ABB sleep mode.

2. AT command operation with standard firmware

Our boards are shipped from the factory programmed with a FreeCalypso firmware image that provides standard end user AT modem functionality with a full commercial product level of stability; at the present time this stable production firmware is FC Magnetite built in the *hybrid* configuration, featuring our new TCS3-based full source version of the G23M protocol stack, replacing the old blob version from Openmoko.

To use the FCDEV3B as a standard AT modem, connect the power supply, one or both UARTs (only UART 0 is needed for this mode of operation) and the antenna, insert a valid SIM with an associated active service from your local GSM network operator into the SIM socket, and press the PWON button or short JP1. The green LED will light up, the modem firmware will boot, and a working AT command interface will appear on the MODEM UART. You can then use your choice of terminal program on whatever host computer you are using to drive the FCDEV3B to talk standard AT commands to the modem; the default baud rate is 115200 bps and you should have hardware RTS/CTS flow control enabled. (If the host system you are using has no RTS/CTS flow control capability and you leave the CTS_MODEM signal on J301 unconnected, the pull-down resistor on the FCDEV3B will present a CTS-asserted state to the Calypso UART, and the AT command interface will work correctly as long as your host does not miss any output from the modem.)

2.1. Effect of deep sleep on the AT command interface

When you start familiarizing yourself with the AT command interface on our current V2 boards using an interactive terminal program, you will immediately notice an effect of sleep modes: when you type your commands manually, one character at a time at typical human typing speed, the very first character you type after power-up or after a period of idleness (10 s or more) will be ‘‘swallowed’’, and then the modem will wake up and accept your subsequent typing, for as long as the delay between successive characters does not exceed 10 s — and if you type nothing for 10 s, the modem goes back to sleep and needs to be awakened again with a ‘‘throwaway’’ press of the carriage return key.

This seemingly-erratic behaviour of the AT command interface is an effect of a power saving mode called deep sleep, and this deep sleep mode is necessary in order to reduce the modem's power consumption during idle times to an absolute minimum, producing the long battery life which mobile phone users have rightfully come to expect. However, this deep sleep mode also has a user-visible effect on the AT command interface: the main 26 MHz VCXO is stopped during deep sleep, leaving only a 32.768 kHz clock, and the latter clock is not fast enough to allow Calypso UARTs to register incoming characters. If the external host sends some command to the modem on the AT command UART channel while the modem is in deep sleep, a special asynchronous circuit in the Calypso will detect the presence of ''some activity'' on the UART data line and initiate the wake-up sequence, but the process of waking up from deep sleep takes about 55 ms, and all characters sent to the modem during this time will be lost.

As often happens in all of engineering, there are no perfect solutions, only trade-offs and compromises, and this case is no exception: the user must choose between either making their modem control software sleep-aware with a special protocol, or disabling deep sleep and accepting greater power consumption during idle states in exchange for host interface simplicity.

2.1.1. Making modem control software sleep-aware

If you are developing your own host system software for talking AT commands to FreeCalypso modems, and you wish to make your modem control software compatible with deep sleep to allow the most power saving, you need to implement the following protocol:

- The first time you send anything to the modem after power-up or long idle, you need to begin by sending a harmless do-nothing command (for example, **`"AT\r"`** or just a carriage return character by itself) which the modem may or may not receive and act upon depending on whether it happens to be awake or asleep at that moment, which you cannot predict or control. After this dummy wake-up command, insert a 60 ms delay to ensure that the modem has had enough time to wake up if it was asleep, and only then send your actual intended command.
- During your ongoing communication with the modem, you need to keep track of how much time has elapsed since your last transmission to the modem on the UART channel. If less than 10 s has elapsed since the last host-to-modem transmission, you can safely send further commands to the modem without additional dummy wake-up commands or delays because the modem is not allowed to enter deep sleep at this time, but if the elapsed time since the last command transmission is 10 s or more, then the modem is allowed to enter deep sleep at unpredictable times, and you need to repeat the wake-up sequence (a do-nothing command followed by a 60 ms delay) before sending the next command. For safety margin, you should set the actual threshold in your software to a little under 10 s, for example, 9.5 s.
- If you are using the GSM 07.10 MUX, the most straightforward wake-up sequence is to send an extra HDLC 0x7E flag (which will do nothing if the modem was awake and therefore receives it), then a 60 ms delay, then whatever actual HDLC frame you were going to send.

Please note that the 10 s UART activity timer in the firmware which prevents the modem from going into deep sleep is only restarted when the host sends something to the modem, and is NOT started or restarted on modem-to-host transmissions. The modem may be configured to send unsolicited notifications to the host in idle mode on cell reselection events or when the received signal strength changes, and by not running the UART activity timer on such events, we avoid going without deep sleep for 10 s every time. The host interface implication is that your software must consider only the time of its last transmission to the modem, and NOT consider the last time it received something.

2.1.2. Disabling deep sleep

If you would rather disable deep sleep to get a quirk-free AT command interface, issue an **AT%SLEEP=5** command. This command will disable deep sleep while keeping the other two sleep modes (big and small sleep) which do not stop the 26 MHz VCXO and therefore have no effect on the AT command interface. Because the modem is initially in deep sleep upon power-up, you need to first send a dummy carriage return, wait 60 ms, and then send the actual **`"AT%SLEEP=5\r"`** command — but you only need to do it once at boot time, and then the AT command interface becomes free of any timing quirks.

To restore the default sleep mode configuration with all 3 sleep modes enabled, issue an **AT%SLEEP=4** command. This sleep mode configuration setting is strictly volatile and does NOT persist across power cycles or reboots.

2.1.3. Board revision differences

This entire section discussing deep sleep and how to either deal with it or disable it is applicable only to our current FCDEV3B V2 boards. Our first FCDEV3B V1 boards (the ones that were sent out to crowdfunding contributors in 2017) have a hardware design defect (an incompatibility between TI's reference design and the reset timing requirements of the high-capacity Spansion flash chip that was only discovered after the boards were built) that requires disabling all sleep modes and not just deep sleep. Our production firmware releases prior to 2019 (the ones that were shipped with those V1 boards) were built in the no sleep configuration (all sleep modes disabled by default on boot), and we also have a special *nosleep* build of our current 2019-04-09 fw version to allow these V1 boards to run up-to-date firmware.

Issuing an **AT%SLEEP=***n* command with *n* equal to anything other than 0 (disable all sleep modes) on an FCDEV3B V1 board (any fw version) will re-enable sleep modes, but doing so will result in unstable operation with random hangs or spontaneous self-reboots resulting from the flash chip being subjected to improper reset timing as the Calypso goes into and out of small sleep.

2.2. Connecting to a GSM network

To bring up the SIM and radio interfaces and connect to your local GSM network, your AT session should begin with the following commands:

AT+CMEE=2 −− enable verbose error messages
AT+CFUN=1 −− enable SIM and radio interfaces
AT+COPS=0 −− connect to the default cellular operator

Our TI-based AT modem implementation differs from general industry practices in that our modem does not automatically turn on its radio and connect to the first available GSM network immediately on power-up, instead it must be explicitly commanded to do so with **AT+CFUN=1** and **AT+COPS**. This quirk of our modem behaviour is not an original FreeCalypso invention, instead we have inherited it from TI and Openmoko. Openmoko's embedded GSM modem in their Neo 1973 and Neo FreeRunner smartphones worked the same way, but Openmoko hadn't invented it either: we have one of TI's original development boards (called D-Sample) that came with a firmware image (pure TI) built in 2002, and that ancient fw also works the same way. We have no plans to change this behaviour in FreeCalypso: even though it differs from general industry practices, our way gives more control to the user.

You can also sandwich additional configuration commands between **AT+CFUN=1** and **AT+COPS**: for example, **AT+CREG=1** to make the modem report registration status changes or **AT+CNMI=2,1** to get notifications of incoming SMS.

2.3. Exercising voice and CSD calls

Once you are connected to a GSM network as above, both voice and CSD calls can be dialed with the classic **ATD** command: **ATD***number***;** (with semicolon) for voice or **ATD***number* (no semicolon) for CSD. Incoming calls can be answered with **ATA**.

If you are going to be exercising voice calls as opposed to CSD, you will need to issue some additional non-standard AT commands of our own invention to enable the right voice interface (see §1.3):

- If you wish to use the on-board loudspeaker driver to hear the call downlink audio, you need to enable it as follows:

  AT@SPKR=1

  The default ABB volume control setting is −6 dB; to set the maximum volume of 0 dB, issue the following command:

AT+CLVL=255

- If you wish to use the digital voice PCM interface on MCSI pins (see Appendix B), you need to switch the voice path selector in our firmware as follows:

  AT@VSEL=1

  This **AT@VSEL=1** command needs to be issued at the beginning of your modem session, before the first call, and it persists for the lifetime of the session.

2.4. Sending and receiving SMS

Our TI-based FreeCalypso modem supports text and PDU modes for SMS as specified in GSM 07.05. Standard AT command **AT+CMGF** selects between these two modes, and as required by the spec, PDU mode is the default. Text mode (which needs to be selected with **AT+CMGF=1**) is intended to be usable directly by a human user from a dumb terminal without any special SMS software, but it has one major limitation: only plain, non-concatenated (no UDH) SMS in GSM 03.38 7-bit encoding can be sent and received in GSM 07.05 text mode in a directly human-readable form. As soon as someone sends you a concatenated SMS or any SMS encoded in UCS-2, a standards-compliant AT modem has no choice but to present it in raw hex form, at which point all direct human readability goes out the window. Because you can only control your outgoing SMS and cannot practically control what kind of SMS other people will send you, practical SMS usage requires special SMS software that communicates with the AT modem in GSM 07.05 PDU mode.

Many FOSS programs have been written over the years to send and receive SMS via an AT command phone or modem; because each vendor's AT command implementation has its own quirks and because we did not want to be dependent on someone else's software written for some other phone or modem manufacturer's quirks, we have developed our own SMS tools (called FreeCalypso User Phone Tools) which officially work with our FreeCalypso AT modems. FC User Phone Tools are included as part of our FC host tools package.

2.4.1. Bogus programming by SIM card issuers

If you still wish to use SMS manually in text mode without special software despite the limitations of this mode, please be aware that some SIM card issuers have misprogrammed their cards in a way that breaks SMS sending and writing operations in text mode. Please look at the description of the **AT+CSMP** command in the GSM 07.05 spec: the settings manipulated by this command control SMS sending and writing operations in text mode, and these settings are stored on the SIM. A reasonable setting would be **AT+CSMP=1,,0,0**, or perhaps a variant with some validity period setting, but one Austrian prepaid SIM card issuer has been caught selling SIMs with PID and DCS parameters set to 255, which is completely bogus and breaks SMS sending and writing operations in text mode. Manually setting **AT+CSMP=1,,0,0** clears the breakage.

2.5. Exercising GPRS

Our TI-based FreeCalypso modem fully supports GPRS in addition to GSM. Our GPRS implementation has been tested by connecting to the mobile Internet service of our local operator (T-Mobile USA) via an FCDEV3B modem with the following sequence of steps:

1. Configure the APN with **AT+CGDCONT**
2. Establish the data connection with **ATD*99***1#**
3. Run Linux pppd

The above procedure has been successful with the versions of the Linux kernel and pppd that came with Slackware 13.37, but has not yet been successful with Slackware 14.2 — more investigative work is needed for the latter.

2.6. Other AT modem functionality

Our modem implementation which we've inherited from TI also includes support for supplementary services, fax calls, GSM 07.10 MUX and a lot of other functionality which we have not thoroughly exercised yet. Please feel free to play with all of these functions; the source code for our modem firmware would be your best

guide as to what is there to be played with.

2.7. Hardware and firmware identification

Our modem implements standard **AT+CG***xx* commands for manufacturer, model and revision identification; the strings returned by these commands are not hard-coded in our firmware, but are programmed in the modem's flash file system at the factory (see §3.4), allowing product identification beyond the firmware. A side effect of this design decision is that these standard **AT+CG***xx* commands identify only the hardware and not the firmware, and **AT+CGMR** in particular gives only the hardware revision. Getting the firmware version requires using a non-standard AT command that was originally added by TI and subsequently modified for FreeCalypso.

You can query the running firmware for its version ID string with the **AT%VER** command; the same version ID string is also emitted as part of the boot-time debug trace output on the other UART (see Chapter 3) and can be seen in the firmware binary image with *strings*(1). This version ID string includes the firmware family name (currently Magnetite), the build configuration (currently *hybrid*, previously *l1reconst*), the corresponding source version and the build timestamp.

2.7.1. FreeCalypso IMEIs

Per GSM standards, every phone or modem that can connect to public GSM networks must have an IMEI number that uniquely and unambiguously identifies the device make and model (the first 8 digits) and the individual physical device unit (the complete number). As the manufacturer of FreeCalypso hardware products, we have diligently gone through the official process for IMEI range allocation, and we now have an officially allocated TAC (the first 8 digits of the IMEI) that officially belongs to our FreeCalypso FCDEV3B product. Each individual FreeCalypso device unit sold by our company is shipped with a fully legitimate and world-unique IMEI number assigned out of our officially allocated range, and our devices are thus fully fit for end user operation on public GSM networks.

Because our FreeCalypso IMEIs are now fully legitimate, you can stand tall and proud before your local GSM network operator as a user of the world's first cellular modem product whose manufacturer officially publishes its complete implementation source code and encourages users to study and understand it: if a GSM network operator looks up the IMEI of the connected device in the official database to see what kind of device it is, they will see that it is a FreeCalypso FCDEV3B modem, made by Falconia Partners LLC, with Mother Mychaela listed as the technical contact.

To query your FreeCalypso modem for its IMEI, issue an **AT+CGSN** command. This command will return the IMEI as a 15-digit string in the standard format, consisting of 14 content digits followed by a computed Luhn check digit. Alternatively, you can issue an **ATD*#06#** command, which will return a 17-digit string: 14 content digits of the IMEI, the Luhn check digit on those IMEI digits, and 2 digits of SV. The SV digits are supposed to identify the software version, and the assignment of these digits is an exclusive prerogative of whichever organization owns the TAC in the first 8 digits.

2.8. Battery voltage monitoring

If you are going to power your board from a Li-ion battery cell in combination with an external charger, you will need a way to see the state of charge of your battery, so you know when it is time to recharge it. Because pure modems like FCDEV3B (as opposed to complete phone handsets) have no built-in battery charging circuits, our firmware for the modem configuration does not include our FCHG battery charging and monitoring driver. We do, however, include a simple custom **AT%VBAT** command that returns the battery voltage in mV, as measured by the MADC block in the Iota ABB.

2.9. Cleanly ending an AT modem session

If you wish to cleanly shut down the modem's activities on the GSM network and tell the network that you are signing off (IMSI detach), issue an **AT+CFUN=0** command. Once you have cleanly disconnected from the network, you can power the modem off with an **AT@POFF** command — the green LED on the board will go out. However, if JP1 is shorted, a true power-off is not possible and the **AT@POFF** will effect a reboot instead, by way of the VRPC block in the Iota chip going through a switch-off sequence immediately followed

by another switch-on from the grounded PWON input.

Alternatively the **AT@RST** command can be used to effect a firmware reboot by way of a watchdog reset irrespective of whether JP1 is shorted or not.

2.9.1. Power-off by button

With recent FreeCalypso fw versions it is also possible to command a power-off (equivalent to the **AT@POFF** command) by pressing the PWON button, holding it down for 1 s, and then releasing it. (This provision was originally implemented back at TI, but it was broken for non-UI modem configurations until we fixed it.) Naturally this option is not possible if PWON is permanently grounded with a shorting block on JP1 — and the latter condition is NOT misinterpreted as a power-off request.

Please note, however, that a power-off by button is strictly equivalent to the low-level **AT@POFF** command: it is a ‘‘raw’’ power-off without an IMSI detach or any other higher-level clean-up. Therefore, it is not a substitute for proper clean-up by external modem control software, instead it allows you to power the modem off without unplugging the power supply if you accidentally powered it on with a button press without a host computer connection. Because our modem does not bring up its radio on its own without a host command, there is no need for higher-level clean-up in the case of an accidental power-on without a host computer connection.

3. FreeCalypso host tools and other firmware versions

Regular operation of a FreeCalypso modem via standard AT commands as described in the previous chapter does not require any special software — one can use any host computer or even a dumb terminal. However, if you have a GNU/Linux PC or laptop or any other Unix-based or Unix-like host system on which you can install FreeCalypso host tools, you get a whole bunch of extra toys to play with:

- A tool for flashing firmware images into the modem, so you can flash new firmware updates or images you have built yourself;
- Tools for decoding and displaying and/or logging the debug trace output from a running firmware, so you can see what the fw is doing when you AT-command it to perform this or that GSM operation;
- Tools for sending debug commands to the running firmware, so you can peek and poke registers and memory locations, enable additional debug traces and much more;
- Tools for manipulating the modem’s flash file system;
- Many more hacks you can discover by studying the source code and docs included in the FC host tools package.

To gain all of these abilities, you need to download and install our FreeCalypso host tools package; the current version as of this writing is fc-host-tools-r10 and the installation procedure involves compiling from source. The rest of this chapter will assume that you have FC host tools installed and working.

It also needs to be noted that our FreeCalypso host tools significantly predate FC hardware, i.e., most of our tools were developed long before we had any hardware of our own, when we were limited to hacking various pre-existing Calypso devices. Keeping this historical fact in mind will make it easier to make sense of much of the documentation included with the tools.

3.1. Reflashing the firmware with fc-loadtool

The utility for performing raw flash operations on FreeCalypso devices is *fc-loadtool*. On sensible devices like our FCDEV3B that have the Calypso boot ROM enabled by the board wiring, *fc-loadtool* can gain flashing access to the device through either of its two UARTs completely irrespective of its previous state, i.e., it is impossible to brick the board no matter what you do to its flash.

Out of the many things you can do with an FCDEV3B board, flashing with *fc-loadtool* is unusual in that it works exactly the same whether you go through UART 0 or UART 1 (`/dev/ttyUSB0` or `/dev/ttyUSB1` in the common FT2232D USB setup). Most other workflows work only with one UART or the other: for the standard ASCII AT command interface with our standard firmwares you need UART 0, and for the RVTMUX binary packet interface (*rvtdump*, *rvinterf*, *fc-shell*, *fc-fsio*, *fc-tmsh*) you need UART 1. But flashing with *fc-loadtool* is the unusual exception in that it works exactly the same through either UART.

If you are seeking to flash a new firmware release from FreeCalypso, the tarball package with the firmware update should contain a file named *flash-script*. It is a command script for *fc-loadtool* that contains the correct sequence of flash erase and flash program commands for flashing the new firmware image. To flash a firmware update using this included script, cd to the directory containing the *flash-script* file and the firmware image it references, and run a command of the following form:

fc-loadtool -h fcfam -B812500 */dev/ttyUSBx* flash-script

The **-B812500** option tells *fc-loadtool* to use the fastest serial baud rate of 812500 bps, which is a GSM-specific baud rate derived from the 13 MHz clock used in the Calypso and other GSM devices. This non-standard (outside of the GSM world) high baud rate is supported by the FT2232D adapter commonly used with FCDEV3B boards, but may not be supported by other kinds of serial hosts. To use the more standard 115200 baud rate, omit the **-B** option:

fc-loadtool -h fcfam */dev/ttyUSBx* flash-script

If you are using the PWON button to power on and boot your board, i.e., if you don't have a jumper on JP1, the most proper way to perform the above flashing operation is to have the board in the ''powered but switched-off'' state, i.e., with the power supply connected but the green LED off, run the *fc-loadtool* command in this state, and as *fc-loadtool* sends its beacons down the serial line, press the PWON button. The Calypso boot process will be interrupted and diverted at the boot ROM, *fc-loadtool* will feed our FreeCalypso *loadagent* to the latter and use this agent to perform the flash operations. The board will be automatically powered off at the end, i.e., the green LED will go out when the operation is finished.

If you need to perform an *fc-loadtool* operation when there is a jumper on JP1 or when the board is in a hung state, run the *fc-loadtool* command and press the RESET button on the board instead of PWON.

It is also possible to use *fc-loadtool* interactively, without running a script:

fc-loadtool -h fcfam */dev/ttyUSBx*

In this mode the tool will stop once it has got *loadagent* running on the target, and present a **loadtool>** prompt to the operator. You can then execute various commands including flash operations interactively; type *help* at the **loadtool>** prompt to get started.

3.2. Playing with the RVTMUX binary packet interface

While UART 0 carries a standard ASCII AT command interface with our firmwares, there is an additional binary packet interface called RVTMUX presented on UART 1. This interface is intended for development and debugging, not for end-use applications, i.e., the latter should NOT depend on having this second UART available, but for developers and tinkerers it enables the following capabilities:

- The firmware continuously emits a fairly large amount of debug trace output on this interface; this debug trace output can be captured, decoded from binary packets into ASCII and displayed and/or logged with our *rvtdump* utility.
- A number of debug, development and test mode commands can be sent to the firmware in the form of RVTMUX binary packets. GPF ''system primitive'' commands can be sent through *fc-shell* and are primarily useful for selectively enabling various more verbose debug traces in the G23M protocol stack. TM and ETM commands can be sent through *fc-tmsh*; many of them invoke L1 and RF test modes which you should **never** invoke unless you wish to operate a rogue transmitter or jammer, but some ETM commands like memory and register peeks and pokes can be useful in ordinary firmware debugging sessions.
- The modem's flash file system can be manipulated with full read and write access over RVTMUX using our *fc-fsio* utility.
- As a new feature added in FreeCalypso, AT commands can also be sent over RVTMUX, appropriately wrapped in RVTMUX binary packets; our *fc-shell* utility performs this feat. This mechanism can only be used for voice and SMS AT commands which do not involve data connections; for CSD and GPRS you need to use the main AT command interface on UART 0.

3.3. Compiling your own firmware versions

If you are interested in tinkering with our FreeCalypso modem firmware and building your own versions from source, with or without your own modifications, you have two different source trees to choose from as your starting point:

- FC Magnetite is our official mainline firmware development tree. Our production fw releases are built from Magnetite, but if you are compiling it yourself from source, you can select many different configurations that differ from our production builds even without changing a single line in the actual source.
- FC Selenite is an experimental firmware source tree that replaces our earlier FC Citrine dead-end code. Like its predecessor, FC Selenite offers the option of building with gcc (instead of TI's proprietary TMS470 compiler used for our stable production builds), and in the gcc-built configuration it is totally free of any blobs. However, this configuration is strictly experimental, not production, and still exhibits a lot of unfixed breakage resulting from the use of a compiler that was never envisioned or considered by the original developers. Fixing all of that breakage and getting Selenite-gcc to work as well as Magnetite is left as a project for any interested community members.

3.3.1. Running firmware out of RAM without flashing

A special feature of our FCDEV3B hardware not present on most of the pre-existing Calypso devices is our large RAM (8 MiB) which allows any of our firmwares to run entirely out of RAM without flashing. The primary intended mode of usage is keep a production firmware image in the flash while running experimental builds out of RAM.

When you build any of our firmwares from source, you can build either a flashable image or a RAM-loadable one. The two image types are not interchangeable: if you have a flashable image (`fwimage.bin` or `fwimage.m0` from a Magnetite or Selenite build or `flashImage.bin` from a Citrine build), you cannot run that same image out of RAM, instead you need to build an appropriate RAM-loadable image: `ramimage.srec` for Magnetite and Selenite or `ramImage.srec` for Citrine. (RAM-loadable images are always in SREC format.)

Once you have a `*.srec` RAM-loadable image, you can load and run it with the *fc-xram* utility. Just like flashing with *fc-loadtool*, the initial steps performed by *fc-xram* work exactly the same whether you go through UART 0 or UART 1, but your choice of UART for this operation needs to be made with the rest of the workflow in mind:

- If you are going to run the XRAM download over UART 0, your *fc-xram* invokation command should take the following form:

  fc-xram -h fcfam */dev/ttyUSBx* ramimage.srec

  The sequence of PWON or RESET manipulations needed to initiate the operation is the same as for flashing with *fc-loadtool*. Once the XRAM image is loaded, it will start running, and if the serial port used by the *fc-xram* process corresponds to UART 0, then the newly loaded firmware's ASCII AT command interface will appear when *fc-xram* drops into the tty pass-through mode. Of course one can also have an *rvinterf* process running in another window, waiting for RVTMUX debug trace output on the other UART.
- If you are going to run the XRAM download over UART 1, your *fc-xram* invokation command should take the following form:

  fc-xram -h fcfam */dev/ttyUSBx* ramimage.srec rvinterf

  Note the addition of *rvinterf* after the `*.srec` image name. This additional command line argument instructs *fc-xram* to pass the serial channel to *rvinterf* when the loaded image starts executing; such passing is appropriate when the XRAM download is done over UART 1, as the newly loaded firmware will immediately start emitting debug traces in RVTMUX binary packet format on this UART.

When run as shown above, *fc-xram* will use the standard 115200 baud rate for the XRAM image transfer. If you are using an FT2232D USB adapter or some other serial host that can support GSM-specific high baud rates, you can use the **-B812500** option to speed up the image transfer over the serial line.

3.4. Flash file system and its content

In addition to the main firmware image which must be reflashed as a single unit when it needs to be changed, the flash memory on the FCDEV3B also holds a file system structure — the FFS. All TI-based GSM mobile station firmwares require a flash file system to be available for both reading and writing in normal operation; the initial creation (formatting) of this FFS is a special operation which TI intended to be performed only once in the lifetime of each individual device, as part of the factory production line procedures.

The makers of various historical Calypso-based devices have used the FFS facility in different ways, and in the FreeCalypso family of projects we have likewise created our own unofficial ‘‘aftermarket’’ FFS configurations when running our firmwares on alien hardware. But on our own FCDEV3B the FFS area of the flash is used as follows:

- Every FCDEV3B unit sold by us has had its radio tract individually calibrated at the factory, and the resulting calibration values are stored in the FFS. These RF calibration tables are required for correct radio operation; if they are missing or corrupted, the modem will usually fail to find a serving cell (frequency burst acquisition will fail) and never reach the point of trying to transmit anything, but if an uncalibrated modem does manage to pick up a cell and starts transmitting (the usual Location Update when registering to a network), its transmissions will be out of spec and may draw the wrong kind of attention from radio regulators.

  It thus follows that the factory-written RF calibration tables in the FFS should be treated as read-only by all ordinary users and firmware tinkerers; recalibration or any changes to these tables should only by done by those who **really** know what they are doing and have the necessary RF test equipment.
- Identification strings naming the hardware manufacturer, the device model and the hardware revision have also been programmed in the FFS at the factory; these strings are returned in response to **AT+CG***xx* queries by the standard firmwares. There is nothing to stop you from changing these strings, but doing so should never be necessary and is discouraged.
- The factory-assigned IMEI (or more precisely, the IMEISV to be used by our official firmwares) is stored in the FFS. Because it is just a regular file, there is nothing to stop you from changing it, but we wholeheartedly ask you to **please** please *please* not do it: now that we have obtained fully official and legitimate IMEIs for our modems and no longer need to borrow numbers from old out-of-business manufacturers' ranges, there is no longer any need to change your IMEI to anything other than our factory-assigned number. We strive very hard to get our FreeCalypso modems accepted as fully legitimate by the mainstream (non-FOSS) GSM community of operators, regulators and related entities, and having users change their IMEIs for no good reason would cause severe harm to the mission of our project.
- The audio mode configuration tables for the **AT@AUL** command are stored in the FFS. We do program these audio mode config files into the FFS of our devices on the production line, but these configurations are expected to change and evolve with new developments just like the firmware, hence FCDEV3B users need to be able to install any potential updates to these files themselves.
- If you wish to play with the ringtone melody generator built into the Calypso (the Melody E1 function implemented in the DSP ROM code and driven by the firmware's Layer 1 and Audio Service components), you will need to upload the E1-format melody files for it into the FFS before you can play them. The voice memo recording and playback facilities similarly use the FFS.
- There are a bunch of files which our TI-based firmwares write into the FFS on their own in normal operation; you can ignore these files unless you wish to put on the hat of a firmware developer.

The host utility for reading and writing the FFS of FreeCalypso devices is *fc-fsio*. It is an *in vivo* tool rather than an *in vitro* one: it works by communicating with the running firmware on the board through the RVTMUX interface on UART 1 (see §3.2), either by connecting to an already-running *rvinterf* process or by launching its own private instance of *rvinterf* (the actual RVTMUX binary packet communication engine). Please refer to FreeCalypso host tools documentation for usage details.

3.4.1. Updating the audio mode configuration files

The source for the audio mode config files resides in the *fc-audio-config* Mercurial repository. The configuration bits in question need to be compiled from source into binary form before they can be uploaded into the FFS of an FCDEV3B or other FreeCalypso device, but the tools needed for this compilation are included in the same FC host tools package as the *fc-fsio* utility you will need for the actual upload, hence no additional dependency is created. Please see the README file inside the *fc-audio-config* repository for further instructions.

3.4.2. FFS corruption, backup and restore

The fact that the flash file system of FreeCalypso GSM devices like the present FCDEV3B contains RF calibration values which cannot be recreated without highly specialized RF test equipment leads to the requirement that this FFS must never get corrupted. TI's FFS implementation has been specifically designed to withstand and gracefully recover from any possible firmware crashes or power cuts without getting corrupted, and while some historical versions may have had bugs in them (the proprietary fw of the Pirelli DP-L10 phone is known to corrupt its FFS on occasion), we are reasonably confident that our version does not suffer from such bugs: we are using the same version of the FFS implementation code as Openmoko, and through all of the years of Openmoko there has not been a single report of the modem's FFS getting corrupted.

There is, however, a difference in that we produce and market our products with the specific intent that they will be extensively tinkered with at the low level, rather than sell locked-down phones or declare the modem to be a ''thou shalt not enter'' forbidden area like Openmoko did. Thus even if we take the stance that none of our officially released firmware versions will ever corrupt the FFS, the fact that our products are specifically intended for low-level tinkering implies that accidental FFS corruption can easily result from operator error on the part of such a low-level tinkerer. Therefore, if you are going to tinker extensively with the flash on your board, you should make a backup of its FFS with the original factory RF calibration values.

To make a backup of the FFS at the raw flash level, run *fc-loadtool* in interactive mode as shown in §3.1. Once you are at the **loadtool>** prompt, issue the following command:

```
flash2 dump2bin my-ffs-backup.bin 0 0x200000
```

To restore an FFS backup made as above, issue the following commands at the **loadtool>** prompt:

```
flash2 erase 0 0x200000
flash2 program-bin 0 my-ffs-backup.bin
```

Finally, if all else fails, you can send your board back to the factory for recalibration — as long as no one abuses it, we should be able to provide this warranty service free of charge except for shipping costs.

4. Operation without FreeCalypso firmware

The FCDEV3B has been created for the primary purpose of running FreeCalypso software and firmware. FC firmware came first, running on various pre-existing Calypso hardware targets with the help of FC host tools, and then we have created our own FCDEV3B hardware in order to free ourselves from the limitations of those pre-existing Calypso devices. However, there exists a certain population who seek to use our hardware for the non-intended and non-approved purpose of running our competitors' OsmocomBB software; this chapter addresses that population.

In the time period between 2017-04 (when our first FCDEV3B boards were assembled and we were doing initial bring-up) and the beginning of 2019-03 we provided a limited degree of support for running OBB software on our hardware, and during that time period we even produced two packages of OBB *layer1* software with our own modifications which sought to improve its radio operation: obb-fcmods-r1.tar.bz2 dated 2017-11-24 and obb-fcmods-r2.tar.bz2 dated 2019-02-25. However, a series of tests performed on 2019-03-09 using our CMU200 RF test instrument revealed that OBB's radio transmissions (with or without our changes, which make only a very small improvement) are very grossly in violation of GSM specifications, and are **highly likely** to cause harmful interference and disruption to GSM or other cellular networks if one were to run this same OBB software on a Calypso GSM MS device (be it our FCDEV3B or an old Motorola phone) with a real antenna connected (as opposed to the cabled connection to the CMU200 instrument in our test setup), transmitting on actual live airwaves.

On the basis of this observation of OBB's gross misbehaviour in its radio transmissions, we are now withdrawing our support for that software effective immediately, until and unless the owners and custodians of OsmocomBB (i.e., those people who believe that it should be continued, maintained and further developed, contrary to our stance that it should instead be retired to the dustbin of history as a bad project) fix its radio transmissions and bring them into compliance. As of this writing, our FCDEV3B modems are safe to use on public airwaves **only** with our official FreeCalypso firmware, and **not** with OsmocomBB.

Anyone who chooses to run OBB software on any Calypso GSM MS device with radio transmissions enabled despite having read this warning will not only be causing interference and disruption to public radio communication networks, but doing so **knowingly and willingly**, which involves a significantly higher level of criminal culpability.

Appendix A: Analog loudspeaker and microphone

TI's Iota ABB chip (part of the Calypso+Iota chipset around which our board is built) is designed to drive 32 Ω earpiece speakers directly, without needing any external active components, but it cannot do the same for 8 Ω hands-free loudspeakers: the ability to drive such loudspeakers directly was added in TI's later chipsets, but is not present in the Iota ABB targeted by FreeCalypso. Instead our FCDEV3B features the same arrangement that was implemented on TI's own D-Sample and Leonardo development boards: the analog voice downlink output from the Iota ABB (from the EARN&EARP pins in our case) goes to an external audio amplifier, and the output of that amplifier is presented on the two-post header at J312, where an 8 Ω loudspeaker needs to be connected. The external amplifier needs to be enabled via Calypso GPIO 1, which is controlled with the **AT@SPKR** command with our standard firmware.

The loudspeaker connected to our board needs to have a load impedance of 8 Ω or greater, with exactly 8 Ω being the standard choice for sufficiently loud acoustic output in a setup where the board and the speaker rest on a lab bench, and the speaker needs to be comfortably heard without having to lift it to one's ear. Any 8 Ω speaker can be used as long as it can handle the drive level put out by our board: the theoretical maximum that our circuit can drive into an 8 Ω load is 0.81 W (with the Calypso DSP programmed to put out a sine wave at full scale on the digital voice interface going to the Iota ABB, and with the volume control in the latter set to the maximum of 0 dB), but for practical uses a 0.5 W speaker rating should be sufficient. However, a physically larger speaker should be selected in order to produce sufficiently loud acoustic output: as we have learned empirically, a cellphone-sized speaker will NOT produce satisfactory output without going through a special mechanical and acoustic design effort as would be done for a real cellphone product, thus for operation on a lab bench without any special acoustic design a larger speaker should be used. We are currently using a 5 cm diameter round speaker (8 Ω, 0.5 W) sold by SparkFun (part number COM-09151), and it produces very good acoustic output in our application.

Our board also features a microphone input circuit copied from TI's Leonardo schematics; the actual microphone needs to be of the electret condenser type, and needs to be connected to two-post header J313; there is a **+** mark on the silk screen indicating the correct polarity. We are not aware of any specific requirements for the microphone part, but we are currently successfully using CUI CMC-9745-130T parts from Digi-Key; this particular part has been selected for its physically larger size, allowing for easier soldering.

To obtain a working loudspeaker+microphone kit for exercising voice calls on your FCDEV3B, you can either buy your own speaker and mic parts and do your own soldering and crimping (the latter is needed in order to produce female connectors that will mate with the two-post headers on our board), or you can buy a fully assembled kit from Falconia Partners LLC. If you are using one of our kits, please be sure to connect the microphone correctly, as it is a polarized component: the little arrow on the female connector in our assembly must face toward the **+** mark on the PCB silk screen.

One remaining issue with the voice subsystem is that the audio configuration settings in the Calypso DSP and in the Iota ABB remain to be tuned for optimal operation. One setting which we already know needs to be tuned is the Acoustic Echo Cancellation (AEC) feature of the DSP. Given that the FCDEV3B is a development board that is expected to rest on a lab bench rather than a handheld device that can be held up to a user's ear, the expectation is that voice call tests will be performed with a loudspeaker that is loud enough to be heard from a comfortable distance away. In this setup the output of this loudspeaker will also get picked up by the microphone, and the party on the other end of the call will hear a delayed echo of their own voice. This problem is not unique to our development board arrangement and also occurs in all standard hands-free phone loudspeaker

setups, and the Calypso DSP includes the AEC feature as the solution. However, this AEC is disabled by default (it is not needed in the classic handheld setup with the earpiece speaker pressed against the user's ear), and needs explicit enabling and configuration. The characterization work to come up with a good AEC configuration has not been done yet; please feel free to beat us to it.

Appendix B: Digital voice interface via MCSI

The Calypso chip features an auxiliary 4-wire synchronous serial interface (data in each direction, bit clock and frame sync) called MCSI. This interface is connected to the DSP part of the Calypso and is controlled by the DSP code (ROM or downloaded patches); the ARM part of the Calypso where regular firmware code runs has no direct control over this interface. The DSP ROM code in the Calypso silicon version we are using supports a mode of operation in which MCSI becomes an external digital voice channel interface; TI called it the Bluetooth mode because they used it in Bluetooth-enabled phones to connect the digital voice channel between the Calypso and their Bluetooth Island chip.

The MCSI signals (4 signal pins plus ground) are brought out to a 5-pin header on the FCDEV3B, allowing our board to be used as a platform for exercising and testing the digital voice interface feature of the Calypso chipset. The MCSI header on the FCDEV3B is J311, and it has the following pinout (pin 1 is toward the edge of the board, away from the power input connector):

1	MCSI_CLK
2	MCSI_RXD
3	MCSI_TXD
4	MCSI_FSYNCH
5	GND

The signals have 2.8 V native logic levels and are tolerant of 3.3 V, but NOT any higher voltages. MCSI_RXD is always an input to the Calypso, MCSI_TXD is always an output from the Calypso, but the direction of the bit clock and frame sync signals is determined by the DSP code: in pure hardware terms they are bidirectional, but given the level of difficulty in developing custom DSP patches, their direction is fixed in practice by the DSP code in use. The DSP code configures MCSI_CLK and MCSI_FSYNCH to be outputs when the interface operates as a ''Bluetooth'' digital voice channel, i.e., the Calypso acts as a PCM master on this interface and cannot be made into a PCM slave except via a custom DSP patch.

The format in which digital voice samples are exchanged via MCSI between the Calypso and the user's application processor is 16-bit linear PCM, 8000 samples per second, thus requiring 128 kbps of bandwidth. The synchronous serial interface details are as follows:

- When MCSI is enabled, the Calypso puts out a 520 kHz clock on MCSI_CLK, produced by dividing the 13 MHz master clock by 25. This clock as put out by the Calypso has a perfect 50% duty cycle.
- Given that a new pair of samples (uplink and downlink) needs to be transferred once every 125 µs (1/8000th of a second, for 8000 samples per second), this 520 kHz bit clock is further divided by 65 to produce an 8 kHz clock on MCSI_FSYNCH, i.e., every 65 bits there is a frame synchronization pulse on MCSI_FSYNCH. The width of this frame sync pulse is one full bit time, the pulse is active high, and the MCSI_FSYNCH line stays low the rest of the time.
- Calypso outputs MCSI_FSYNCH and MCSI_TXD are updated on the rising edge of MCSI_CLK and should be sampled on the falling edge of the same clock by the user's application processor. The MCSI_RXD input is sampled on the falling edge of MCSI_CLK by the Calypso, and should be updated on the rising edge of the same clock by the user's application processor.
- Each voice sample (both uplink and downlink) is transferred from the most significant bit to the least significant bit (MSB first), starting with the clock cycle immediately following the frame sync cycle, i.e., the cycle in which MCSI_FSYNCH is asserted by the Calypso.
- Given that each frame is 65 bits long (520 kHz bit clock divided by 8 kHz frame rate) and that 16 bit slots out of every frame are used to transfer voice sample bits, plus one bit slot for frame sync, it follows that 48 bit slots out of every frame are dummies. The Calypso puts out 0 bits on MCSI_TXD in all bit slots that don't carry voice sample bits.

At the present time there does not exist any off-the-shelf hardware for connecting a PCM master interface such as ours (most other vendors' GSM and newer cellular modems also provide very similar PCM interfaces for digital voice) to a general-purpose computer. Aside from probing the interface with an oscilloscope or a logic analyser to verify that it works as expected, we are considering building the following adapters for our MCSI:

1) We are investigating the possibility of connecting our MCSI (through a custom level-shifting adapter) to an McBSP interface on a classic OMAP board called BeagleBoard-xM, producing a proof of concept for how a Neo900-style smartphone can be built with a FreeCalypso modem.

2) If the goal is to interface our digital voice channel to a general-purpose computer (PC or laptop) as opposed to building a smartphone prototype, the ideal gadget would be an adapter that would turn our PCM interface into a USB device, presenting a standard USB sound card to the host. Because no one makes such a gadget to our knowledge, we would have to design and build it ourselves. One promising possibility would be to use an Atmel AT91SAM7S chip, which has a USB device controller, a synchronous serial controller, and an ARM7TDMI processor (same as Calypso) for the firmware. The design can be prototyped on an off-the-shelf AT91SAM7S-EK evaluation board, and then a smaller custom board can be made.

It needs to be noted, however, that both of the above ideas are currently at a very low priority relative to other FreeCalypso work.

Appendix C: Calypso JTAG

The JTAG header on the FCDEV3B is J310, and it has the following pinout (pin 1 is toward the power input connector):

TMS	1	2	nTESTRESET
TDI	3	4	GND
Vio	5	6	NC
TDO	7	8	GND
TCK return	9	10	GND
TCK	11	12	GND
EMU0	13	14	EMU1

All JTAG signals have 2.8 V native logic levels and are tolerant of 3.3 V, but NOT any higher voltages. The JTAG interface is needed very rarely; it is not needed at all in normal usage and even during intensive firmware development a need for JTAG arises only very rarely.

C.1. Unconventional reset

The JTAG interface provided on Calypso platforms like our FCDEV3B (same as on TI's own historical development boards) has one major difference from ''classic'' ARM JTAG in terms of the reset logic, a difference which you MUST thoroughly understand if you wish to play with this JTAG interface using OpenOCD or similar tools. The difference is that the Calypso chip does NOT have a TRST pin, and instead the reset signal provided on JTAG connector pin 2 (which is defined as TRST on more conventional platforms) is more like SRST — and even then the analogy isn't great.

There is only *one* reset signal available in the Calypso+Iota chipset that can be asserted from the outside, and this signal is nTESTRESET, implemented in the VRPC block in the Iota ABB. It is an extremely deep reset, and it resets *everything*, even the RTC domain, just as if battery power to the chipset were completely removed and then restored. It is not possible for any state anywhere, particularly in the Calypso JTAG logic, to survive across nTESTRESET: the Calypso JTAG interface (of which nTESTRESET is NOT a part!) lives in the Vio power domain, cannot function until the Vio regulator is turned on, and must be hard-reset when Vio may have gone away, whereas nTESTRESET is a function of VRPC, a block that controls all of those regulators and which itself functions when all of them are off.

On the FCDEV3B this nTESTRESET debug reset can be triggered in two ways: by pressing the RESET button (it simply shorts nTESTRESET to GND), or by driving a logic low into JTAG connector pin 2 from an external adapter. However, this JTAG connector pin 2 is not ''raw'' nTESTRESET — the latter is referenced

and pulled up to a non-logic voltage rail and cannot be safely connected to external logic like JTAG adapters. Instead there is a transistor circuit on the board, copied from TI's Leonardo schematics, that allows an external JTAG adapter to assert nTESTRESET without connecting it directly. The effect of this circuit is that if the external adapter drives a logic low into this pin (either from an OC/OD driver or from a conventional push-pull kind), the internal nTESTRESET will be asserted. Thus it is safe to connect our JTAG connector pin 2 to either kind of driver, which would not have been the case for ''raw'' nTESTRESET.

C.1.1. CPU halt directly out of reset

One very commonly desired function in the world of JTAG-based debugging is to halt a CPU directly out of reset, i.e., have the CPU reset, but instead of executing instructions from the reset vector, hold still in the JTAG debug state. Many processors have reset logic with separate TRST and SRST, with SRST resetting everything except JTAG logic which is reset only by TRST; on such processors one can enter the debug halt state through the JTAG scan chain, then assert SRST, and get the desired effect. But this approach is not possible on the Calypso: our chipset has *only one* reset signal, and this very deep reset blows away everything, including all JTAG state.

Instead the Calypso chip most definitely does have an ability to have the ARM7 core hold still directly out of reset, and so did TI's earlier DBB chips that had no internal boot ROM, but it is invoked in a very non-standard manner for which we still don't know the full details and which remains to be reverse-engineered. It is my (Mother Mychaela's) educated guess that this function is probably invoked via one of the two EMU pins, which are completely undocumented except for the line entries in the pinout table which say that they are bidirectional (IN/OUT) signals with internal pull-ups.

To reverse-engineer this function, someone will need to figure out which XDS hardware version and which CCS software version was used at TI to operate on Calypso platforms Back In The Day, get that historical XDS+CCS combination up and running, connect it to our FCDEV3B (which should work as it is exactly the same as TI's own D-Sample board in this regard), and reverse from there.

C.2. Buffered vs. unbuffered FTDI-based adapters

In the hobbyist tinkerer community the most popular kind of JTAG adapters are those based on an FT2232C/D or FT2232H chip (or perhaps FT232H or FT4232H) operating in FTDI's MPSSE mode. These adapters fall into two kinds: they can be buffered or unbuffered. If you take a generic FT2232x breakout board (e.g., the FT2232D adapter we typically use for our dual UART) and connect its ADBUS pins directly to target JTAG signals, the result will be an unbuffered adapter, whereas a buffered adapter is one in which one or more buffer ICs sit in between the FT2232x and the JTAG target connection interface. An unbuffered adapter can be a generic FT2232x breakout board that is not officially designated as a JTAG adapter at all, whereas a buffered adapter has to be made specifically for JTAG applications — but it should be noted that some ''official'' JTAG adapters can also be unbuffered — Openmoko's debug board is one known offender.

Unbuffered FT2232x adapters are perfectly fine for UART-only (single or dual UART) applications, but for JTAG a buffered adapter is strongly preferable, for two reasons:

1) If a JTAG adapter needs to work with target logic voltages outside of the range directly supported by FTDI (being able to work with 1.8 V targets is the canonical use case), the only way to provide such wide target logic voltage support is to insert a logic voltage level translating buffer.
2) A nasty design misfeature of all FTDI chips with MPSSE support is that this MPSSE mode cannot be configured in the EEPROM, instead it can only be entered by host software command, and until the user runs an appropriate userspace program like OpenOCD, that intended-for-JTAG FTDI channel operates in the power-up default UART mode. If this operating-as-UART channel is connected directly to JTAG target signals, problems can occur.

In the case of our Calypso JTAG interface the logic voltage level problem does not occur (Calypso is 2.8 V native and perfectly compatible with 3.3 V external logic), but the other problem of power-up default UART mode is very real and quite serious. If one were to connect an unbuffered FTDI adapter (such as the FT2232D adapter we use for our dual UART, or an Openmoko debug board) to our Calypso JTAG interface, then for as long as the FTDI adapter is powered up but not switched into MPSSE (JTAG) mode by a userspace program, the FTDI adapter's ADBUS2 driver (an output in the power-up default UART mode) will fight with the Calypso's

TDO driver (always an output from the Calypso), potentially damaging either or both chips. One could work around by powering the FTDI adapter (connecting it to the USB host) with the target interface disconnected, then running the userspace program, then connecting the FCDEV3B — but this method is very prone to human error, and it would be very easy to unplug and replug USB (putting the adapter into the dangerous UART mode) while the FCDEV3B is connected.

For this reason, if you are going to use an FTDI-based JTAG adapter with our FCDEV3B, that adapter needs to be buffered. The Mother of FreeCalypso is currently working on the design of our own FreeCalypso UART+JTAG Adapter (FC-UJA) that will do exactly what we need: it will be an FT2232D adapter with Channel A wired for JTAG with appropriate buffering logic, and with Channel B wired as a generic UART. Our FC-UJA will also be very specific to the Calypso application, i.e., it is not intended to compete in the general-purpose JTAG adapter market: the target interface logic voltage will be fixed at 2.8 V (Calypso native), and there will be just one reset signal driver for our nTESTRESET, as opposed to ‘‘classic’’ TRST and SRST drivers.

For those who cannot wait for our FC-UJA to become available and who need some working JTAG adapter right now, the only existing adapter which we can officially recommend at the present moment is the original Flyswatter (**not** Flyswatter2!) by Tin Can Tools, which is unfortunately out of production and will probably be difficult to find. We cannot, however, endorse Tin Can Tools' current Flyswatter2 product because TCT do not publish the schematics for their current version, unlike the now-discontinued original Flyswatter whose schematics are published. In the absence of schematics it is impossible to tell for certain whether or not a given adapter is safe to connect to our FCDEV3B, and in which specific way it should be connected. We also consider it morally wrong to give our business to a vendor who puts their ‘‘intellectual property’’ or trade secret interests above the safety of their customers.

Once again, this problem will be solved properly once and for all when we have our FC-UJA; until then, if someone needs to use JTAG right now and cannot find an original Flyswatter, please contact Mother Mychaela — we will look for a solution on a case-by-case basis.