

ARDUINO TELEPHONE CALLER ID DEVELOPMENT KIT

Version 1.00 23-Oct-2019

Author: 6V6GT

1 INTRODUCTION

This is a relatively straight forward but comprehensive development system for creating the basis for landline (fixed telephone network) caller id applications such as display of incoming caller number and details, storage of call data, spam call protection etc. from the data transmitted along with each telephone call during the ringing phase. It supports the main protocols for data transmission (MDMF and SDMF) but not the less used DTMF and other signaling methods like interrupted dial tone etc.

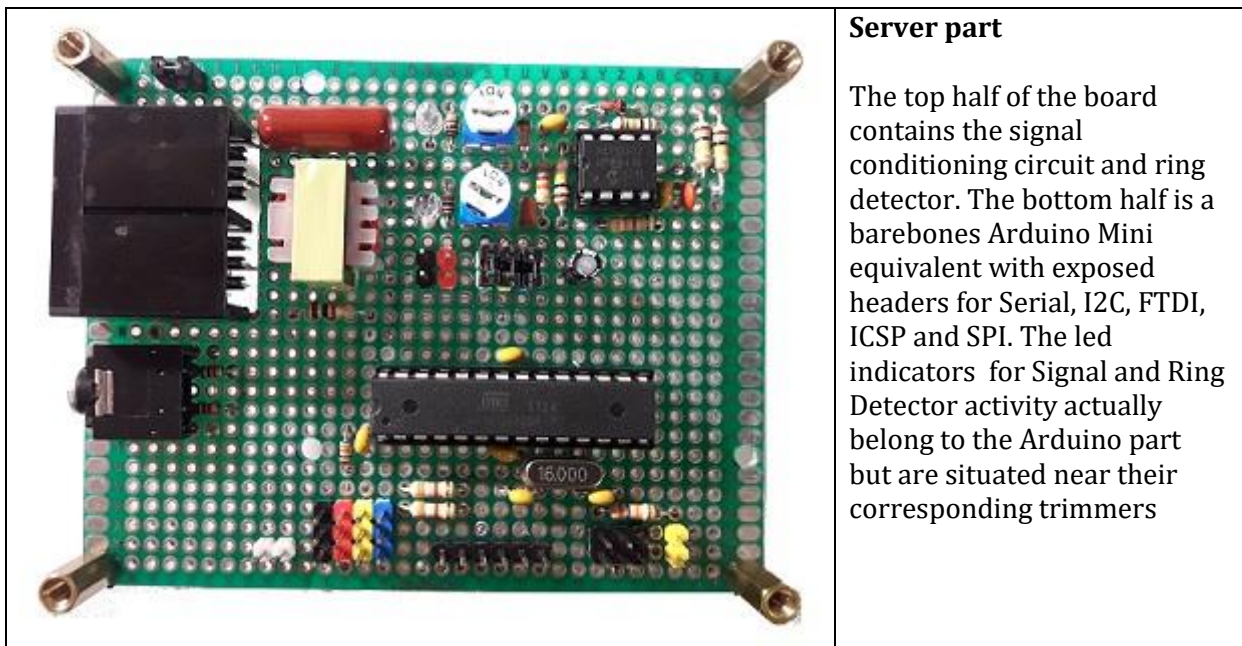
It consists of a reference server based on an ATmega328P with some signal conditioning hardware which demodulates and parses the encoded data, a number of alternative example clients demonstrating different communications methods with the sever (SPI, Serial, I2C) and using different hardware platforms (Windows, ESP32, Raspberry Pi, AVR and Texas Instruments MSP432P401R). For simple applications, the server can even be used stand-alone without a client part, but then care is required when integrating code with it to avoid resource clashes with the signal processing functions. The sample clients have limited functionality, but enough to demonstrate transfer of call data etc. from the server part. Also referenced is a Java script based application for generating test data (incoming call, message waiting etc.) so the system can be tested independently from a telephone installation (published separately – [Ref11]). This solution uses easily available components and does not rely on any telephone specific chips.

Splitting the application over a server part and a client part has a number of advantages, but mainly that the user can develop an application incrementally and including features like call storage, telephone book lookup, special display logic etc. etc. without having to touch the core code for demodulating and parsing the transmitted data and this is a classic "offloading" scenario. Further, the demodulator is quite an intense activity and can be impacted by other activities e.g. continually scanning a touch screen or a web server checking for client connections etc.

Another advantage this offloading brings is that some of the platforms either have no ADC or one of insufficient performance, so since another supporting chip is required for a solution anyway, so why not use one which is programmable and that includes a strong ADC. Incidentally, this is one of the few areas where the ATmega328P is a more useful chip than, for example, the ESP8266 due to the performance of its peripheral devices (or maybe just unavailability of sufficient supporting documentation to exploit the full performance of such features).

The project is also interesting in that it is an illustration of the some of the possibilities for interfacing between an AVR Arduino and a variety of different clients using several protocols.

I have published a couple of complete designs in the past for caller ID systems (see appendix [Ref9] and [Ref10]), but these were not really beginners projects. This is different in that it could be tackled by a beginner but using the server in stand alone mode, say with the optional LCD and omitting any of the client options. However, it stops at the point where it delivers decoded data. The user must then write routines for presentation and, if required, transmission, storage and retrieval of the data.



General disclaimer:

1. Before connecting anything, including this device, to your telephone network, be sure that it does not conflict with any local requirements or requirements of your telephone service provider regarding construction, materials, electrical safety, emission of electromagnetic radiation, certification or other such matters.
2. This development has been tested only in a Swiss network. Although the standards are in principle international, there are differences which may impact the correct functioning in other environments.

2 CONTENTS

1	Introduction.....	1
2.1	Acknowledgements.....	5
2.2	Abbreviations	6
3	System Architecture with alternative Clients.....	7
4	Server	7
4.1	Schematic	7
4.2	Sample layout picture with component labels.	9
4.3	features (inc. debug modes etc.).....	9
4.4	tips on stripping out unused code	10
4.5	Software Description.....	10
4.6	Server/Client Interface.....	11
4.7	Server Configuration / Testing	12
4.8	Connecting Server and Client	12
4.9	Server Software Pack	13
5	Clients.....	13
5.1	General	13
5.2	Reference Client Set (Uno/Nano etc.)	14
5.2.1	Serial.....	14
5.2.2	I2C.....	15
5.2.3	SPI.....	16
5.3	Raspberry Pi 3B + ...	18
5.4	ESP32	20
5.5	Windows Powershell	21
5.6	Texas Instruments MSP432P401R.....	23
5.7	Client Software Pack.....	25
6	AFSK Data Formats	26
6.1	Call data	26
6.2	Message Waiting	27
7	Trouble Shooting	28
8	Appendix.....	28
8.1	Software Pack Description.....	28
8.2	A document collection, some directly referred to above or used in the solution and some generally relevant further reading.....	29

2.1 ACKNOWLEDGEMENTS

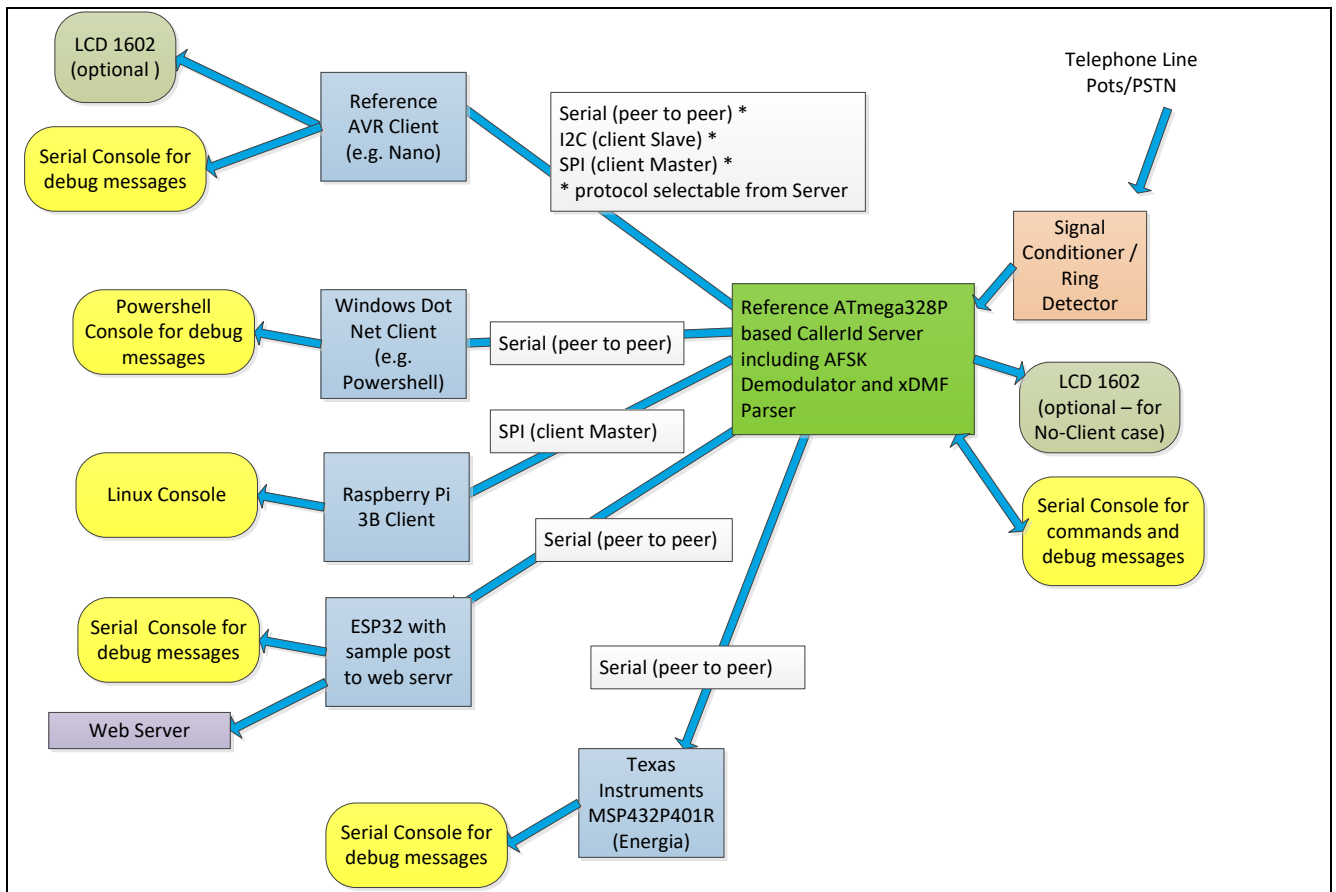
- The software AFSK demodulator is adapted from an APRS modem designed by markqvist (www.unsigned.io)

2.2 ABBREVIATIONS

ADC	Analog to digital converter
AP Mode	Access point mode. The ESP8266 creates an access point which wireless devices can connect to. It is used for an initial configuration.
APRS	Automatic Packet Reporting System
AFSK	Audio Frequency Shift Keying
CLIP .	Caller Line Information Presentation
E164	Telephone number representation e.g. +412071838750
Eeprom	Electrically erasable read only memory
ESP8266	A microcomputer architecture compatible with Arduino.
ETSI	European Telecommunications Standards Institute
IC	Integrated Circuit
ISR	Interrupt Service Routine
MCU	Microcontroller Unit
MDMF	Multiple Data Message Format
POTS	Plain Old Telephony System
PSTN	Public Switched Telephone Network
SDMF	Single Data Message Format (not currently supported here)
xAMP	A server architecture consisting of a mySql database, Apache web server and a PHP as a server side scripting language.
xDMF	SDMF or MDMF depending on the context

3 SYSTEM ARCHITECTURE WITH ALTERNATIVE CLIENTS

The system consists of a server part which handles the interpretation of the AFSK signaling and the ring detection and hands this data to the client part which can handle a basic display and any optional features which the user choses to build. The diagram shows all the optional client types included in the software package, special example features and protocols supported.



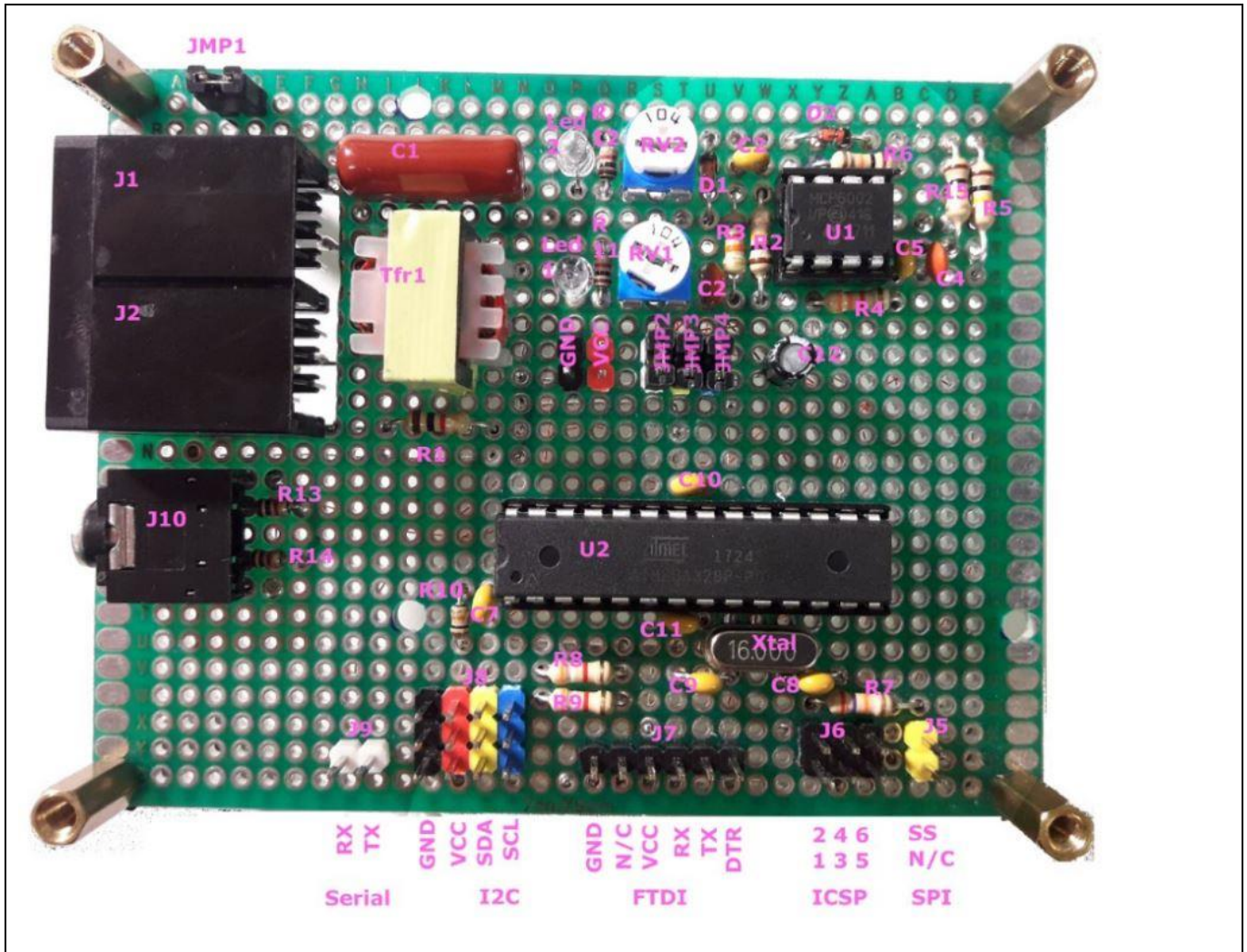
Note:

- (1) All possible client types are illustrated. In the simplest case, the server can operate stand alone with an LCD display.
- (2) The Reference AVR client is 3 separate programs, one per protocol type.
- (3) The Windows Dot Net client / Powershell also include a sample post to a web server

4 SERVER

4.1 SCHEMATIC

4.2 SAMPLE LAYOUT PICTURE WITH COMPONENT LABELS.



4.3 FEATURES (INC. DEBUG MODES ETC.)

Supports Bellcore 202 and ETSI V23 AFSK signaling.

Supports Ring detection (17 to 70 Hz above voice voltage levels)

Includes an MDMF and SDMF parser for caller ID and message waiting data

Diagnostics messages are returned to the client for debugging failed transmissions.

Debug modes are selectable through the server serial console and appear on the servers serial console (if attached)

- <db0> debug mode 0 – no debugging
- <db1> debug mode 1 – demodulator bit stream dump (no client active in this mode)
- <db2> debug mode 2 – general debug level (default)
- <db3> debug mode 3 – enhanced detail.

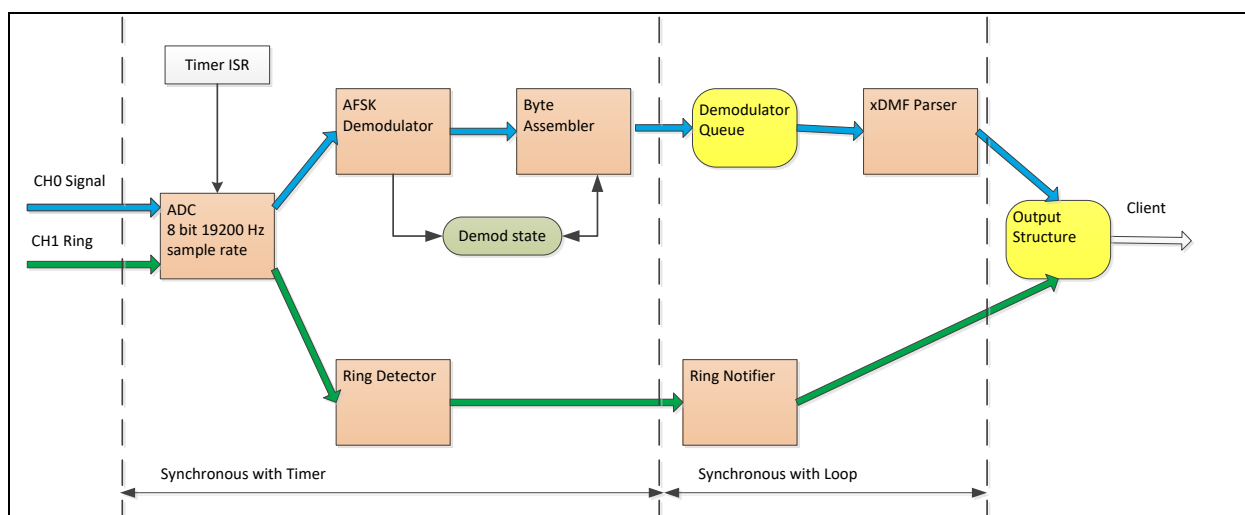
Protocols for communication with the client are selectable via the server serial console:

- <ser> serial communication (peer to peer) default - (protocol #2 in server power on message)
- <i2c> I2C communication (server is master) default - (protocol #3 in server power on message)
- <spi> SPI communication (Server is SPI slave and must be regularly polled by the client)- (protocol #4 in server power on message)
- <ntr> No protocol active (use if no client active) - (protocol #0 in server power on message)

4.4 TIPS ON STRIPPING OUT UNUSED CODE

In almost all cases, a single protocol between the server part and the client part is sufficient. The code has been logically grouped so the code for specific protocols can be isolated. In the simplest case, the hard coded default can be changed in `setup()`. Alternatively, all the code can for the unwanted protocols can be removed. Beware of impacting the debug options by removing the `receiveSerialEvent( )` function. For setup and debugging, it is important to retain this.

4.5 SOFTWARE DESCRIPTION



The ADC sampling rate is controlled by a timer. 8 bit resolution is sufficient for this application. The AFSK demodulator is driven by the ADC ISR and handles the decoding of the input signal into 0s and 1s, stripping start and stop bits, dropping bit stuffing and handling endian

conversion. It recognizes the channel seizure and mark block which precede the data stream and strips these out. The remaining datastream is passed to the byte assembler. This recognizes the packet (or envelope) and processes the header, length and trailer bytes. The remaining bytes are the payload and are queued for processing by the xDMF parser then dumped into the output structure for transmission to the client.

The ring detector uses a software frequency detection to determine the ring status. The input signal to the ring detector is above the voltage level of the voice/AFSK channel so it is not interfered with. The ring status is then dumped into the output structure for transmission to the client.

4.6 SERVER/CLIENT INTERFACE

- 4 structures
- One way
- Data types across clients (endianness & size & word boundaries)
- Diag Description. (diag info passed to client because server console may not be visible)

```

struct Tfr_call_t {
    uint8_t rType ; // 'C' ; // C=Call
    uint8_t msgType ;
    uint8_t checksumStatus ;
    char date[ 9 ] ; // allow 1 for end marker
    char number[ 21 ] ; // ditto
    char name[ 51 ] ; // ditto
} ;

struct Tfr_message_t {
    uint8_t rType ; // 'M' ; // M = Message
    uint8_t msgType ;
    uint8_t checksumStatus ;
    uint8_t mwIndicator ;
    uint8_t mwMsgQty ;
} ;

struct Tfr_diag_t {
    uint8_t rType ; // 'D' ; // Diagnostic message (in case of failure)
    uint8_t msgType ;
    uint8_t checksumStatus ;
    uint8_t dataPacketLength ;
    uint8_t errorCode ;
    uint8_t diagValue ;
};

struct Tfr_ring_t {
    uint8_t rType ; // R = Ring
    uint8_t inLongRingBurst ; // treat as bool
    uint8_t inShortRingBurst ; // treat as bool
} ;

```

msgType is the original header ID in SMDF or MDMF

4.7 SERVER CONFIGURATION / TESTING

There are two potentiometers to adjust. RV1 controls the gain of the AFSK signal detection part and RV2 controls the ring detector part.

For the AFSK signal detection part it is best to put the server in debug mode 1 via the server console command <db1> (see earlier in this chapter) with the device connected to a signal source. That signal source could be a live telephone connection, a connection to a PC sound card etc. while playing an audio sample of an AFSK signal (an audio file is included in the software pack) or the AFSK generator tool described in the following chapter. The AFSK generator tool documentation includes some tips on connecting a PC to the server part. The output from debug mode 1 is usually a fairly clean stream of 0s to the serial console when there is no signal. When a correctly formed signal is detected, recognizable patterns will be seen as illustrated in the chapter describing the data formats. Setting the potentiometer midway between the extremes where the signal is no longer recognizable is a good guide. If you have an oscilloscope, you can check the signal at the output of the op-amp (jmp3 – see schematic). Note, however, that the parser delivers good results even when the signal shows moderate clipping. LED1 (see schematic) will also indicate if a signal has been detected. On a live telephone connection, the signal is delivered (normally) between the first and second ring burst.

For the ring detector part, this is best tested with a live telephone connection and the potentiometer RV2 adjusted according to LED2 indicator (see schematic).

Once the leds show that the device is capable of receiving an AFSK signal / Ring, then you also have the option of further testing by selecting one of the other server debug modes or connecting a 1602 type lcd display to the server part.

4.8 CONNECTING SERVER AND CLIENT

Connect server to USBTTL device via TX, RX and ground only. Power the server via the Client to avoid possible overvoltage on client pins in the case of a 3.3 volt only client. The USBTTL device is required only for configuration and testing of the server part after which the selected client protocol will be fixed. The last used protocol type is stored in the eeprom and recovered on server restart.

Where the client is 3.3 volt, this means a slight overclocking of the server's ATmega328p chip. This is generally not a problem and several Arduino Uno clones have a switchable 3.3 volt mode. However, some form of level shifter would, in such a case, be the optimum solution.

4.9 SERVER SOFTWARE PACK

Sub Directory	File Name	Description
ATmega328p_CallerID Server	AVR_CallerID_v0_41p.ino plus supporting .cpp/.h files	

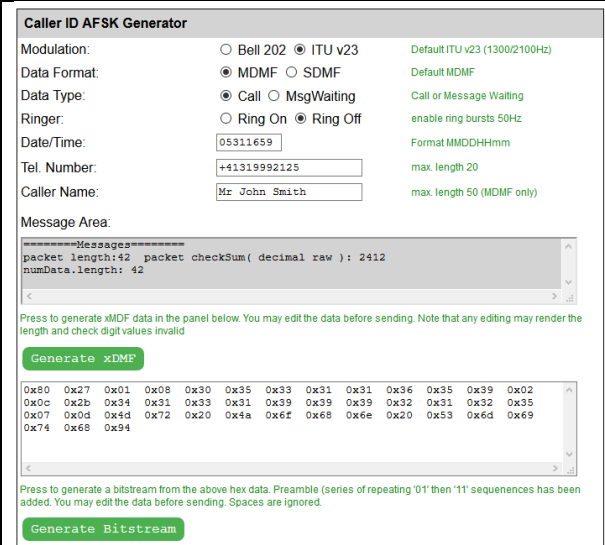
5 CLIENTS

5.1 GENERAL

In principle, the clients obtain data from the server according to the formats defined in the previous chapter. Some clients use the record type identifier and some use the length of the transmitted data to determine the record type (call, message waiting, ring burst or diagnostic message). In the case of Serial and I2C transmissions, the server sends the data to the client. In the case of SPI, the client is the SPI master and checks the server at regular intervals for new data and collects this as required. I2C call data transmission lengths exceed the standard I2C buffer size and so are split into two parts.

The following is a description of each client together with some of the test results. The software for each client is in the accompanying software pack. The screen shots from the used in the client tests is not the current version and there are minor differences to the server start up messages.

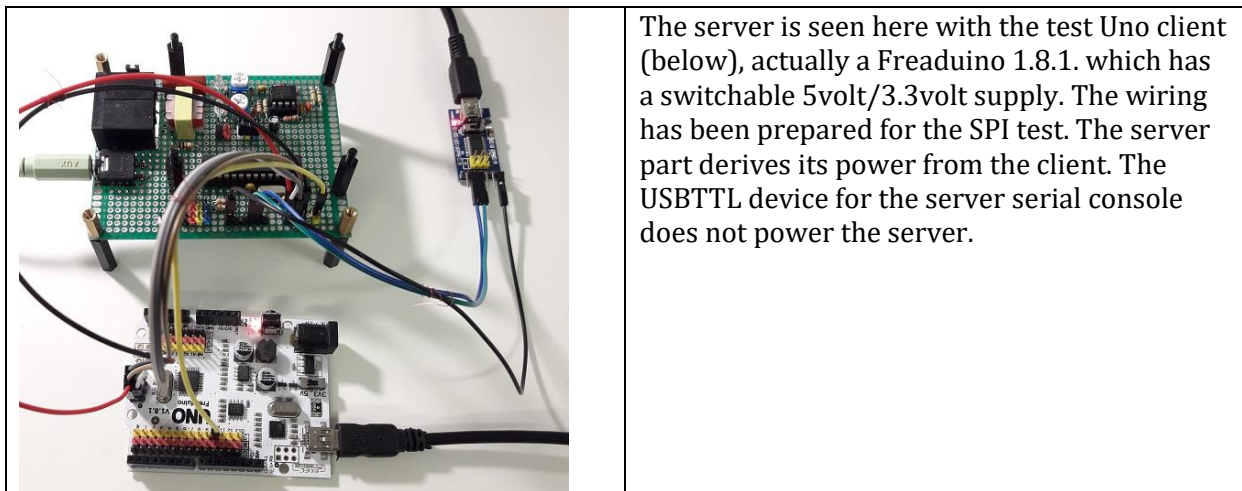
The test results were generated using an audio file (sample also in the software pack) of call meta data in MDMF format including a simulated ring burst or directly using the tool Telephone Caller ID AFSK generator ([Ref11] in appendix)

	A part view of the Telephone Caller ID AFSK generator tool used for server configuration and client testing. It can be used to generate an audio file to simplify further testing.
---	---

5.2 REFERENCE CLIENT SET (UNO/NANO ETC.)

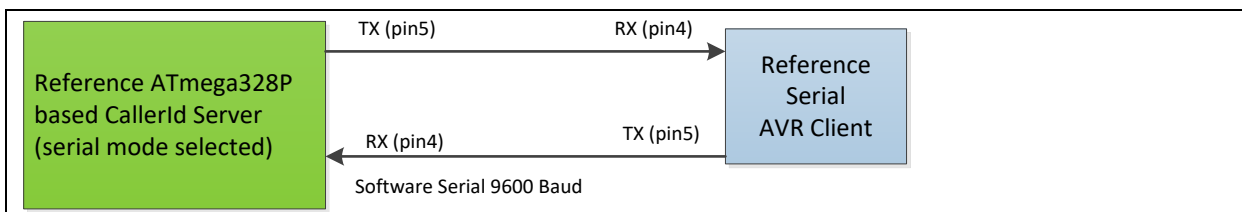
This is a collection of 3 simple clients each demonstrating one of the following protocols: (1) Serial, (2) SPI (client as master) and (3) I2C (client as slave). Each client can receive a full data set from the server representing (a) a call, (b) a message waiting notification, (c) a diagnostic message and a (d) ring indication.

A simple Uno clone (Freaduino 1.8.1) in 5Volt mode was used for these client tests and the Eclipse (Sloeber Plugin) serial console



5.2.1 SERIAL

The server must be switched into serial mode using the server console command <ser> or changing a parameter in the software.



The serial consoles look like this for a sample call.

```
Caller ID version: ACId V0_40
debugMode: 2
Ready to receive commands on Serial or I2C/42
Ready to send data to I2C/25
Transfer Method=2
Run number=293
```

```
version: CID_328ref_Serial v0_01
```

```
>>Send to COM15: "<ser>"<<
```

```

<ser>
transfer method = Serial
++ pkt hdr
++ Checksum OK
>> in processNewInput()

++ call.msgType (HEX): 80
++ call.checkSumStatus (BIN): 1

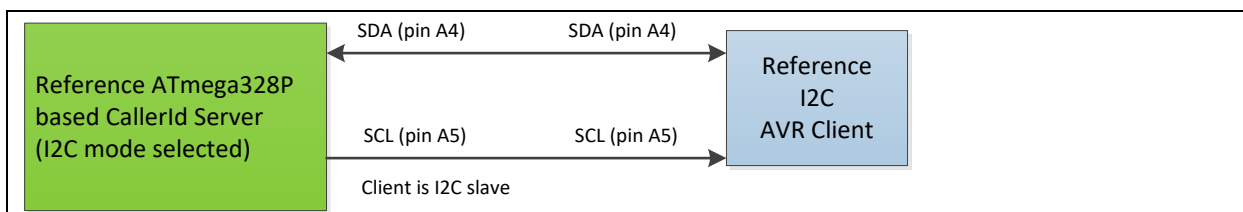
++ call.date: 05311659
++ call.number: +41319992125
++ call.name: Mr John Smith
receiveSerialEvent() first char = C
call:
record header=67
data header=128
check sum=1
date=05311659
name=Mr John Smith
number=+41319992125

```

The black text is from the serial monitor connected to the server part (COM15 in this case). The red text is from the serial monitor connected to the client part (here COM11). These are interleaved. First, the server is set to transmit data in serial format to the client. This is by the command <ser> on the server's serial console. The sample call data is processed by the server part and transmitted over serial to the client part.

5.2.2 I2C

The server must be switched into I2C mode using the server console command <i2c> or changing a parameter in the software.



The serial consoles look like this for a sample call. This sample includes the client receiving a ring burst. The description of the output is otherwise similar to the description for serial above, but with I2C used instead for the communications protocol.

```

Caller ID version: ACId V0_40
debugMode: 2
Ready to receive commands on Serial or I2C/42
Ready to send data to I2C/25
Transfer Method=2
Run number=321

version: CID_328ref_I2C_v0_02

```

```

>>Send to COM15: "<i2c>"<<
<i2c>
transfer method = I2C
ring:
record header=82
inLongRingBurstr=1
inShortRingBurstr=1

>> in processNewInput()

++ call.msgType (HEX): 0
++ call.checkSumStatus (BIN): 0

short ringburst = 1
long ringburst = 1
++ pkt hdr
++ Checksum OK
>> in processNewInput()

++ call.msgType (HEX): 80
++ call.checkSumStatus (BIN): 1

++ call.date: 05311659
++ call.number: +41319992125
++ call.name: Mr John Smith
call:
record header=67
data header=128
check sum=1
date=05311659
name=Mr John Smith
number=+41319992125

ring:
record header=82
inLongRingBurstr=1
inShortRingBurstr=0

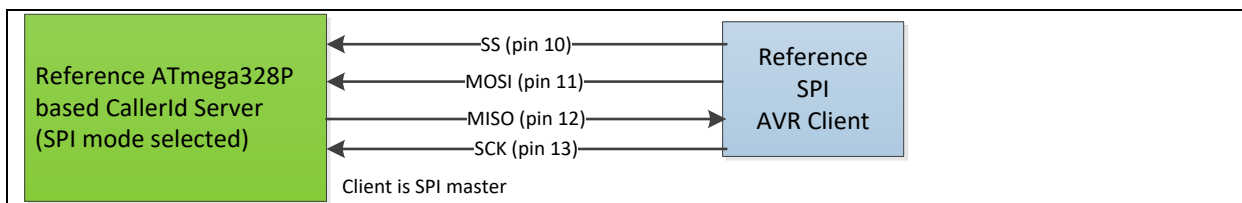
>> in processNewInput()
. . . . .
. . . . .

```

The output has been truncated here to avoid repeating detail.

5.2.3 SPI

The server must be switched into SPI mode using the server console command <spi> or changing a parameter in the software. Note that in this case, the client is the master and collects data from the server.



The serial consoles look like this for a sample call. This sample includes the client receiving a ring burst. The description of the output is otherwise similar to the description for serial above, but with SPI used instead for the communications protocol.

```
Caller ID version: ACId V0_40
debugMode: 2
Ready to receive commands on Serial or I2C/42
Ready to send data to I2C/25
Transfer Method=2
Run number=322
```

```
version: CID_328ref_SPI_v0_01
```

```
>>Send to COM15: "<spi>"<<
<spi>
transfer method = SPI
>> in processNewInput()
```

```
++ call.msgType (HEX): 0
++ call.checkSumStatus (BIN): 0
```

```
short ringburst = 1
long ringburst = 1
```

```
ring:
record header=82
inLongRingBurstr=1
inShortRingBurstr=1
```

```
++ pkt hdr
++ Checksum OK
>> in processNewInput()
```

```
++ call.msgType (HEX): 80
++ call.checkSumStatus (BIN): 1
```

```
++ call.date: 05311659
++ call.number: +41319992125
++ call.name: Mr John Smith
>> in processNewInput()
```

```
++ call.msgType (HEX): 80
++ call.checkSumStatus (BIN): 1
```

```
short ringburst = 0
long ringburst = 1
```

```
call:
record header=67
data header=128
check sum=1
date=05311659
name=Mr John Smith
number=+41319992125
```

```
ring:
record header=82
inLongRingBurstr=1
inShortRingBurstr=0
```

```
>> in processNewInput()
```

```

++ call.msgType (HEX): 80
++ call.checkSumStatus (BIN): 1

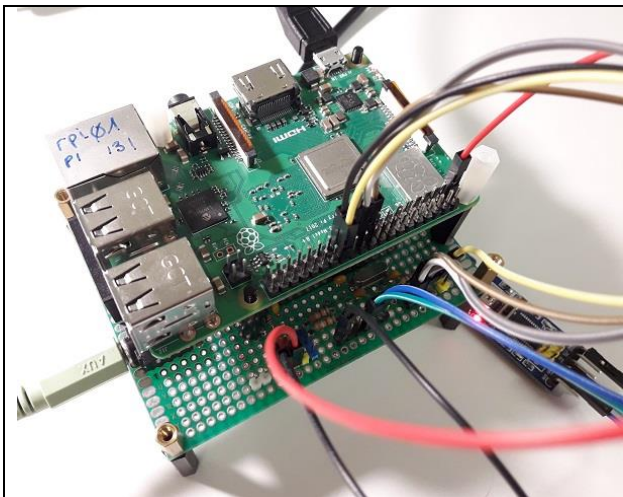
short ringburst = 1
. . . . .
. . . . .

```

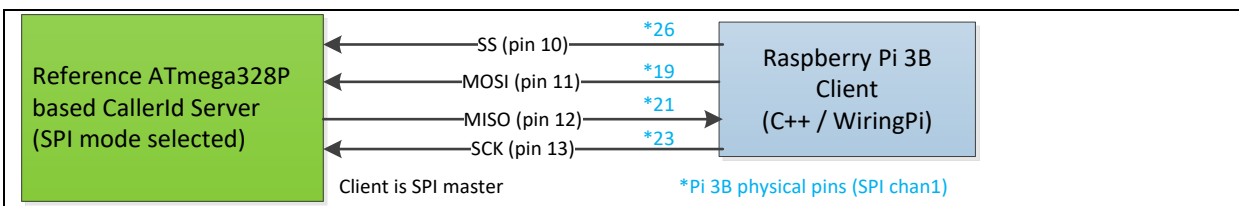
The output has been truncated here to avoid repeating detail.

5.3 RASPBERRY PI 3B +

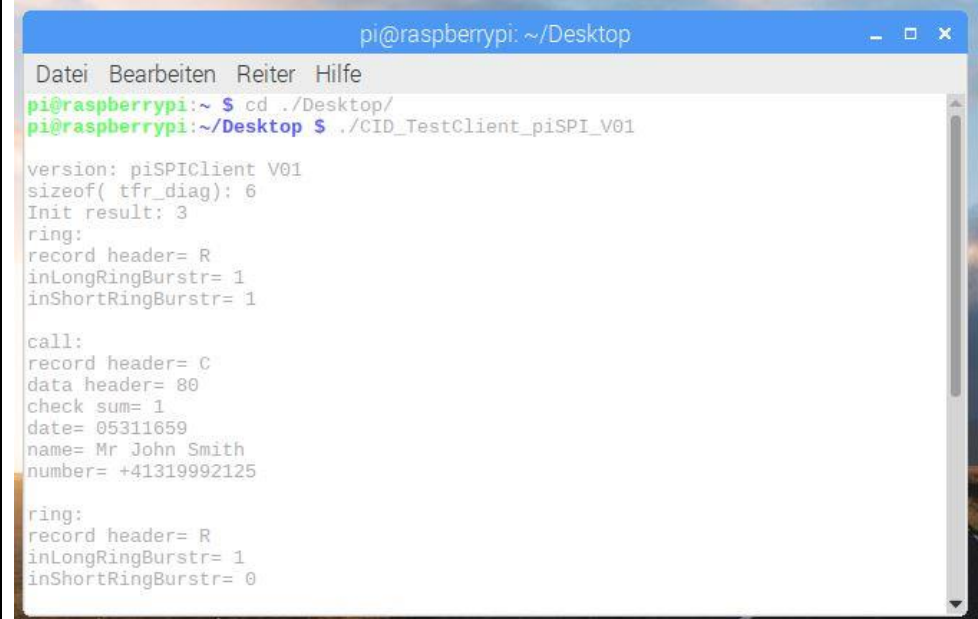
For this client demonstration, a Raspberry Pi 3B + has been configured using NOOBS 3.0.0 with Raspbian as the O.S. For simplicity of use without a dedicated keyboard and screen, a remote desktop client package xrdp has been installed on the Pi so it is contactable over a Windows 10 remote desktop protocol (rdp) session using WLAN. This is slow especially if graphics are displayed, but OK for small amounts of text work such as compiling the client software on the Pi and showing results in its serial console. To support the C++ SPI client software, the wiringPi package has also been installed. For details about configuring a Pi for C++ and SPI see the excellent Sparkfun guide [Ref12]. The Raspberry Pi is a 3.3 volt device so it powers the server and the USBTTL device connecting the server to the PC uses only the RX, TX and ground connections.



The Raspberry Pi 3B+ is illustrated here as a daughter board to the server part. Power for the server is derived from the Pi's 3 volt supply. The USBTTL device for accessing the server serial console via a PC can just be seen towards the bottom right of the picture.



The following is a screen shot of the Pi serial console when receiving data from the server part. The corresponding server serial console appears below it. Both show truncated results for this illustration.



```

pi@raspberrypi: ~/Desktop
Datei Bearbeiten Reiter Hilfe
pi@raspberrypi:~ $ cd ./Desktop/
pi@raspberrypi:~/Desktop $ ./CID_TestClient_piSPI_V01
version: piSPIClient V01
sizeof( tfr_diag): 6
Init result: 3
ring:
record header= R
inLongRingBurstr= 1
inShortRingBurstr= 1

call:
record header= C
data header= 80
check sum= 1
date= 05311659
name= Mr John Smith
number= +41319992125

ring:
record header= R
inLongRingBurstr= 1
inShortRingBurstr= 0

```

```

Caller ID version: ACId V0_40
debugMode: 2
Ready to receive commands on Serial or I2C/42
Ready to send data to I2C/25
Transfer Method=2
Run number=328

```

```

>>Send to COM15: "<spi>"<<
<spi>
transfer method = SPI
>> in processNewInput()

++ call.msgType (HEX): 0
++ call.checkSumStatus (BIN): 0

short ringburst = 1
long ringburst = 1
++ pkt hdr
++ Checksum OK
>> in processNewInput()

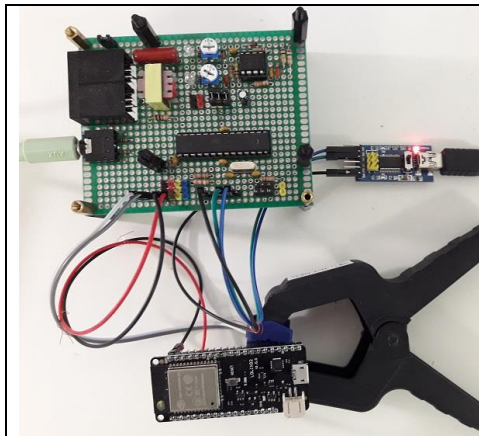
++ call.msgType (HEX): 80
++ call.checkSumStatus (BIN): 1

++ call.date: 05311659
++ call.number: +41319992125
++ call.name: Mr John Smith
>> in processNewInput()
. . . . .
. . . . .

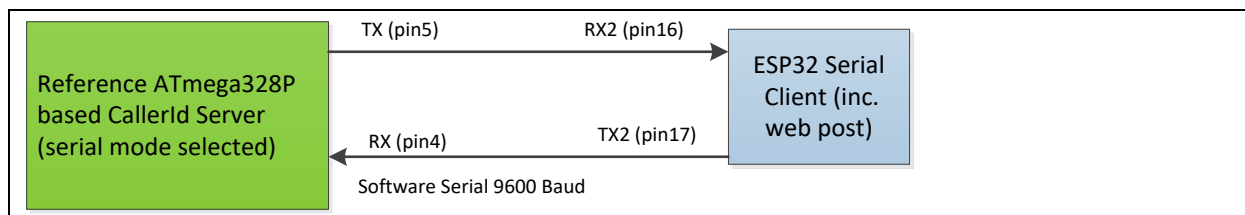
```

5.4 ESP32

For this client demonstration, an ESP32 (Lolin32 v1.0.0) is used in serial mode via Serial2 (RX=16, TX=17). Also included in the test client is sample code to post call data to an external web server. If required, sample code for handling the web PHP and MySQL db can be found in [Ref10]



The ESP32 is a 3.3 volt device. The server is powered by the ESP32 at this voltage. The USBTTL device is switched to 3.3 volts and only the following connections are made from it to the server: RX, TX and ground. USB cable to ESP32 is not shown in this picture.



Interleaved console outputs. Black is the server, red is the esp32 client.

```
Connect to serial port COM15 at 115200
```

```
>>Send to COM15: "<ser>"<<
<ser>
transfer method = Serial
```

```
Connect to serial port COM21 at 115200
```

```
++ pkt hdr
++ Checksum OK
>> in processNewInput()

++ call.msgType (HEX): 80
++ call.checkSumStatus (BIN): 1
```

```

++ call.date: 05311659
++ call.number: +41319992125
++ call.name: Mr John Smith
receiveSerialEvent() first char = C
call:
record header=67
data header=128
check sum=1
date=05311659
name=Mr John Smith
number=+41319992125

http://www.example.ch/myDir/storeCallDataTestN.php?telNo=%2B41319992125&nameNet=Mr
%20John%20Smith&dateStampNet=05311659&checkDigitOk=1&numberInRun=99&runNumber=99
200
telNo: +41319992125nameNet: Mr John SmithdateStampNet: 05311659checkDigitOk:
1numberInRun: 99runNumber: 99

```

The query result posted to the mySql database:

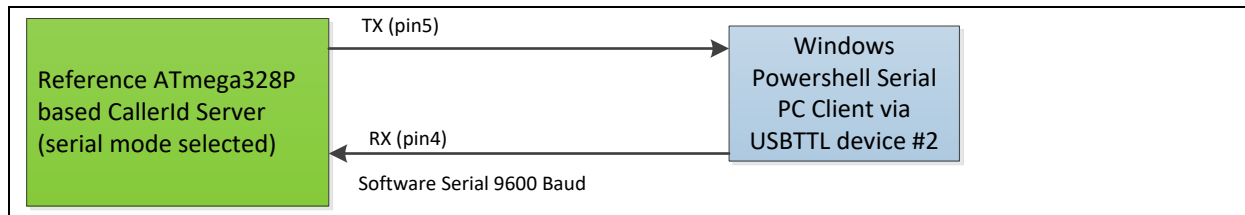
+ Options										
	id	timeStampSystem	telNo	nameNet	nameTelBook	dateStampNet	checkDigitOk	runNumber	numberInRun	
☐ Edit ☐ Copy ☐ Delete	671	2019-06-30 16:08:08	+41319992125	Mr John Smith	NULL	05311659	1	99	99	
☐ Check all With selected: ☐ Edit ☐ Copy ☐ Delete ☐ Export										

5.5 WINDOWS POWERSHELL

This demonstrates interfacing the server part to a Windows .Net PC application. This example uses Powershell version 5. The version level is probably not critical for compatibility with the supplied client code. The source code includes a comment block with usage instructions. You may have to modify one line in the code where the USB Port number is defined. There is an option to dump data to a mySql database through an HTTP GET command to a pre-configured PHP page using the PowerShell `Invoke-WebRequest` verb and this is illustrated in the example below. If required, sample PHP and MySQL code can be found in [Ref10]

If you want to view the server console simultaneously, you will need a second USBTTL device. Ensure that the USB port for USBTTL used by the Powershell client is not attached to any other application e.g. serial console.

No picture of the setup is included here because it is simply the server part with 2 USBTTL devices which are in turn, connected to the PC.



Serial output (truncated) from the server console

```
Caller ID version: ACId V0_40
debugMode: 2
Ready to receive commands on Serial or I2C/42
Ready to send data to I2C/25
Transfer Method=2
Run number=336

>>Send to COM10: "<ser>"<<
<ser>
transfer method = Serial
>> in processNewInput()

++ call.msgType (HEX): 0
++ call.checkSumStatus (BIN): 0

short ringburst = 1
long ringburst = 1
++ pkt hdr
++ Checksum OK
>> in processNewInput()

++ call.msgType (HEX): 80
++ call.checkSumStatus (BIN): 1

++ call.date: 05311659
++ call.number: +41319992125
++ call.name: Mr John Smith
>> in processNewInput()

++ call.msgType (HEX): 80
++ call.checkSumStatus (BIN): 1

short ringburst = 0
```

The results (truncated) in the Powershell ISE

```

Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
cid_ver0.04.ps1 X
1 #
2 # Simple Windows Powershell demonstration of interfacing to an Arduino based Telephone Caller ID
3 # system over a USB serial port (with optional sent to mySql Database via HTTP GET)
4 #
5 # For this simple demonstration, only character information is displayed but all data is transferred
6 # from the server. The first character C|M|R|D indicates how the record should be interpreted.
7 #
8 # code bits from the following and other sources:
9 #
10 # https://blogs.msdn.microsoft.com/powershell/2006/08/31/writing-and-reading-info-from-serial-ports/
11 # https://docs.microsoft.com/en-us/dotnet/api/system.io.ports.view=netframework-4.7.2
12 # https://mcpmag.com/articles/2017/10/19/using-a-stopwatch-in-powershell.aspx
13 #

list of Serial Ports found: COM3 COM10 COM15
port configured by user: COM15
waiting for serial data stream . . .
current_transferType= R
ring
long ring burst= 1
short ring burst= 1
-----
current_transferType= C
call
check sum status= 1
date=05311659
number=+41319992125
name=Mr John Smith

StatusCode      : 200
StatusDescription : OK
Content         : telNo: +41319992125      nameNet: Mr John
                  Smith                      dateStampNet: 05311659checkDigitOk: 1numberInRun:
RawContent      : HTTP/1.1 200 OK
                  Transfer-Encoding: chunked
                  Connection: keep-alive
                  Vary: Accept-Encoding,User-Agent
                  Content-Type: text/html
                  Date: Tue, 02 Jul 2019 17:39:52 GMT
                  Server: nginx
                  X-Powered-By: PHP/5....
Forms           : {}
Headers         : {[Transfer-Encoding, chunked], [Connection, keep-alive], [Vary, Accept-Encoding,User-Agent],
                  [Content-Type, text/html]...}
Images          : {}
InputFields     : {}

Running script / selection. Press Ctrl+Break to stop. Press Ctrl+B to break into debugger. | Ln 61 Col 1 | 125%

```

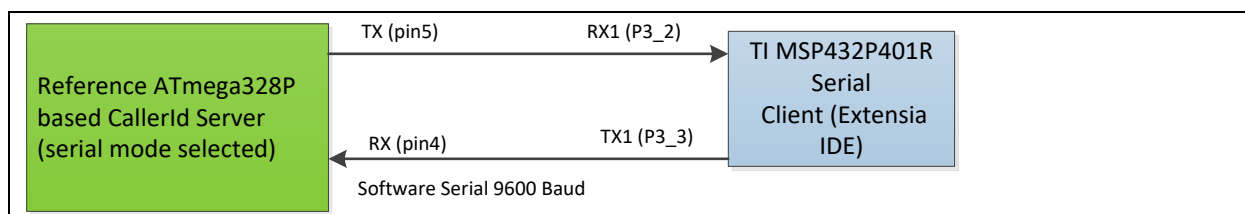
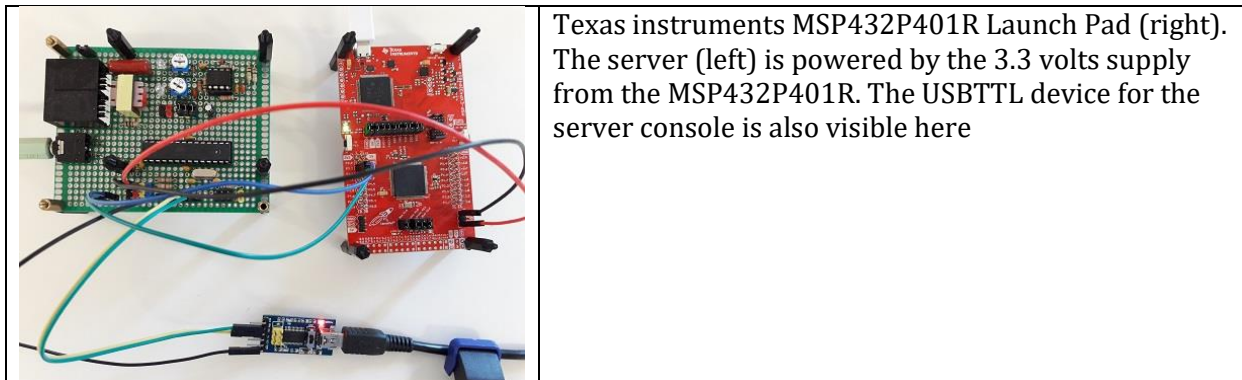
Data dump to mySql database (optional)

+ Options									
←→									
	id	timeStampSystem	telNo	nameNet	nameTelBook	dateStampNet	checkDigitOk	runNumber	numberInRun
☐ Edit ☐ Copy ☐ Delete	675	2019-07-02 19:39:52	+41319992125	Mr John Smith	NULL	05311659	1	0	0
↑ ☐ Check all With selected: ☐ Edit ☐ Copy ☐ Delete ☐ Export									

5.6 TEXAS INSTRUMENTS MSP432P401R

The MSP432P401R is a 32 bit Arm Cortex M4F MCU and is incorporated into the Launch Pad development system used here. The launch pad comes complete with a debugger chip and supported by an online development system CCS cloud, which includes an Eclipse cloud IDE, step through debugger etc. It comes with an Arduino compatible package Energia which allows many Arduino sketches to run unaltered on the MSP432P401R. It also has a Ti version of RTOS for more advanced users.

It is very nice to use but Energia does not have the same degree of community support as Arduino so problem solving is more complicated.



Server console (truncated)

```
Caller ID version: ACId V0_40
debugMode: 2
Ready to receive commands on Serial or I2C/42
Ready to send data to I2C/25
Transfer Method=2
Run number=357
<ser>
transfer method = Serial
>> in processNewInput()

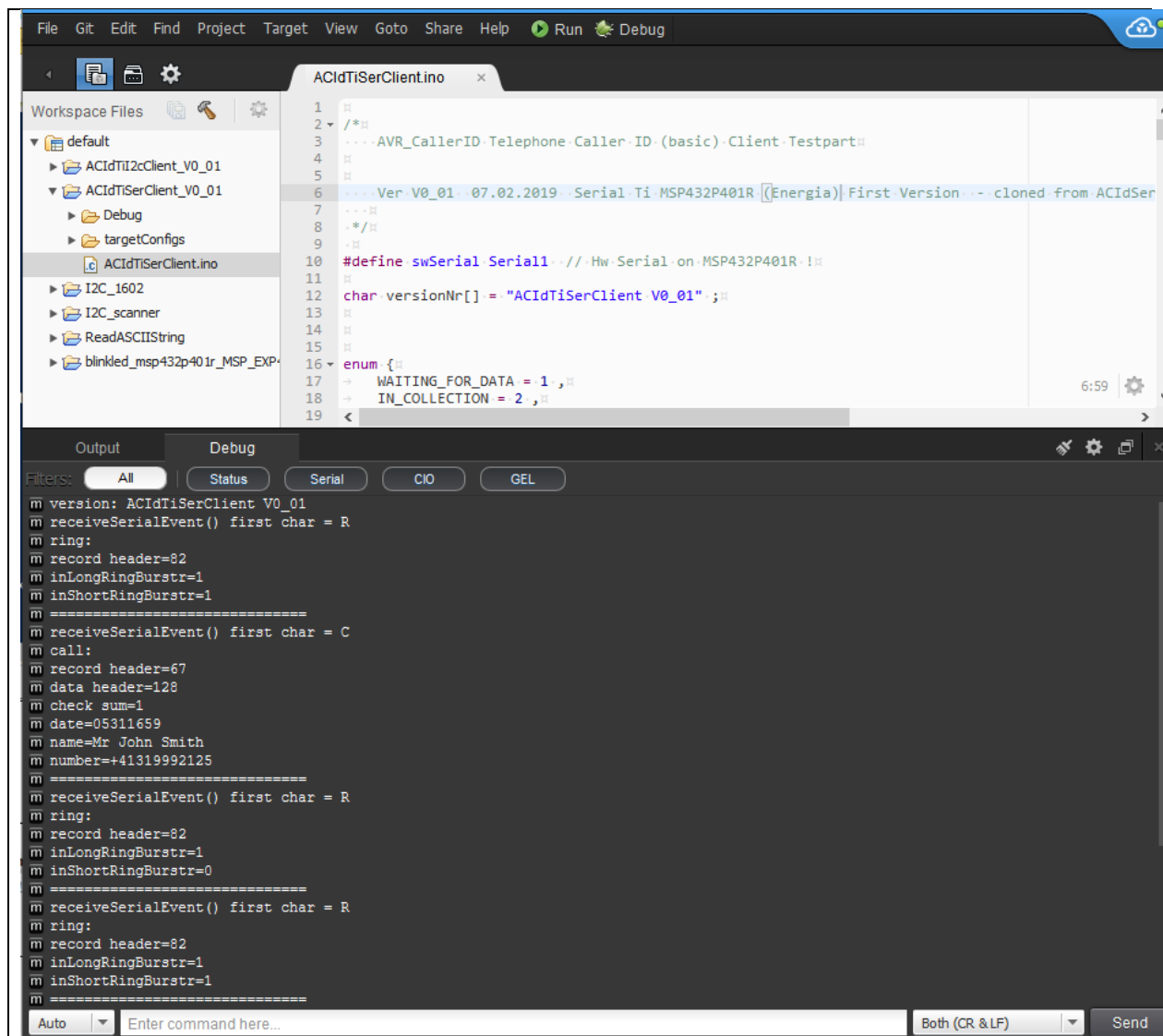
++ call.msgType (HEX): 0
++ call.checkSumStatus (BIN): 0

short ringburst = 1
long ringburst = 1
++ pkt hdr
++ Checksum OK
>> in processNewInput()

++ call.msgType (HEX): 80
++ call.checkSumStatus (BIN): 1

++ call.date: 05311659
++ call.number: +41319992125
++ call.name: Mr John Smith
>> in processNewInput()
```

CCS Cloud online development system with serial console shown.



5.7 CLIENT SOFTWARE PACK

Sub Directory	File Name	Description
ATmega328p_Uno_I2C Client	CID_328ref_I2C_v0_02.ino	
ATmega328p_Uno_Serial Client	CID_328ref_Serial_V0_01.ino	
ATmega328p_Uno_SPI Client	CID_328ref_SPI_v0_01.ino	
ESP32 Client	webClient_V0_03.ino	
RaspberryPi3plus Client	CID_TestClient_piSPI_V01.cpp	
Ti_MSP432P401R Client	ACIdTiSerClient_V0_01.ino	Energia
WindowsPowershell Client	cid_ver0.04.ps1	

6 AFSK DATA FORMATS

The encoding illustrated below is described in several of the quoted references. [Ref5] and [Ref6] are the easiest to follow. The ETSI documents [Ref7] and [Ref8] are the most comprehensive.

The tool Telephone Caller ID AFSK generator [Ref 11] can also be used to create a visual representation of encoded bit stream which could be useful for understanding the format.

6.1 CALL DATA

The best way of describing these is by example, using the output from the application in debug mode 1 which produces an 80 bit wide dump of the signal direct from the demodulator.

[illegible]

The fields between the start(0) and stop(1) marks, some highlighted in purple are the actual call meta data and the first few are decoded below. The ++++++ pattern represents further characters in the data stream which have been obscured here. The parser is fed these bytes and produces the output for the display and storage in Eeprom.

Data Stream value (little endian)	Hex value / character	Description
00000001	0x80	Header byte to indicate MDMF data format
11100100	0x27	Length of following data packet in bytes
10000000	0x01	Date parameter type header marker
00010000	0x08	Length of date parameter in bytes

11010000	0x0B	visual indicator parameter type header marker
10000000	0x01	Length of visual indicator parameter in bytes
11111111	0xFF	Indicator on
11001000	0x13	Number of waiting messages parameter type header marker
10000000	0x01	Length of number of messages parameter in bytes
10000000	0x01	1 message waiting
10001101	0xB1	check sum

The example of no waiting messages is similar except:

1. The visual indicator is off (0x00), not (0xFF)
2. The 3 entries relating to number of waiting messages are not present when the number of messages is zero.

7 TROUBLE SHOOTING

- Beware of the effects of mixing voltages between the server and client parts. Some clients may not tolerate more 3.3 volts on any pin.
- Ensure server is set (via console or by hardcoding) to match the client protocol (serial, I2C, SPI).
- Interpret diag codes on client. To interpret the diag codes, you currently have to look at the server code which generates them (pending document upgrade)
- Keep any mobile phone you are using for testing a few meters away from the circuit to avoid interference.
- If you don't get a fairly clean pattern in debug mode 1 on the serial console when there is no activity on the line, i.e. no ringing, then you have to solve that before proceeding further. The variable resistor RV1 should have an initial setting of around 60k, but adjust as required. Debug is mode 1 selectable via the server console.
- The ring detector should be tuned via the potentiometer so the mark/space ratio is about 50 % and a special sketch may help to make this ratio visible (not supplied). An oscilloscope could also be useful. The ring should be a sine wave, but in reality may be nothing like one. In the worst case, the circuit may have to be tuned to local conditions.
- In standard mode, debugging information is written anyway to the serial console which can be useful if the signal delivered to the MCU is of a reasonable quality.
- There is some fine tuning of the mark and space frequencies available to handle different network standards. See AFSK.h (source software)

8 APPENDIX

8.1 SOFTWARE PACK DESCRIPTION

CallerID_devkit_software_V1_00 containing server, clients and an audio test file.

Location : <https://forum.arduino.cc/index.php?topic=642458.0>

8.2 A DOCUMENT COLLECTION, SOME DIRECTLY REFERRED TO ABOVE OR USED IN THE SOLUTION AND SOME GENERALLY RELEVANT FURTHER READING.

[Ref1] LibAPRS an Arduino soft modem library which provided the basis for the demodulator code.

<http://unsigned.io/projects/libaprs/>

[Ref2] Rane Note: Interfacing Audio and Pots including some basic electrical parameters of the telephone network and an interface circuit.

<http://www.rane.com/note150.html>

[Ref3] Randolph Telecom Inc. AN-4 (Midcom TN #98) LOW COST TELEPHONE LINE INTERFACE (DAA, FXO)

This is a comprehensive description including a specimen circuit for interfacing to POTS telephone networks

<http://www.randolph-telecom.com/articles/AN-4,%20Low%20cost%20telephone%20line%20interface%20DAA,%20FXO.pdf>

[Ref4] Cypress AN2336 - PSoC® 1 - Simplified FSK Detection

This is an example of using a Cypress manufactured chip for FSK signal processing. It is interesting here because it contains a comprehensive theoretical description of the processing techniques which are relevant to the AFSK demodulator in this solution.

<http://www.cypress.com/documentation/application-notes/an2336-psoc-1-simplified-fsk-detection>

[Ref5] holtek Type I caller ID using the HT9032

<http://www.holtek.com.tw/documents/10179/116745/an0053e.pdf>

[Ref6] EXAR TAN008 Designing Caller Identification Delivery Using XR-2211 For U.S.

https://www.exar.com/files/documents/tan_008.pdf

The two references above are similar application notes for obsolete ICs which describe the data formats and parsing of Caller Line Identification information. Exar has also a similar document relevant to British Telecom networks (TAN009)

ETSI documents

[Ref7] ETSI EN 300 659-3 V1.3.1

http://www.etsi.org/deliver/etsi_en/300600_300699/30065903/01.03.01_40/en_30065903v010301o.pdf

[Ref8] ETSI 2 EN 300 659-1 V1.3.1

http://www.etsi.org/deliver/etsi_en/300600_300699/30065901/01.03.01_60/en_30065901v010301p.pdf

The above are very comprehensive documents describing and specifying relevant telephony data formats and encoding standards at various layers.

[Ref9] arduino caller line identification system

<https://forum.arduino.cc/index.php?topic=490392.0>

This is an early stand-alone development with local storage of call data.

[Ref10] arduino caller line identification system with anti-spam function

<http://forum.arduino.cc/index.php?topic=528459.0>

This connects to a mySql database for telephone book information and call history and suppresses unwanted calls.

[Ref11] Telephone Caller ID AFSK generator

<http://forum.arduino.cc/index.php?topic=599571.0>

This is a Java Script application for testing caller ID systems.

[Ref12] Sparkfun guide to using C++ / SPI on a Raspberry Pi.

<https://learn.sparkfun.com/tutorials/raspberry-pi-spi-and-i2c-tutorial/all>

[Ref13] QWEST Message waiting visual indication (SDMF format)

<http://www.centurylink.com/techpub/77335/77335.pdf>