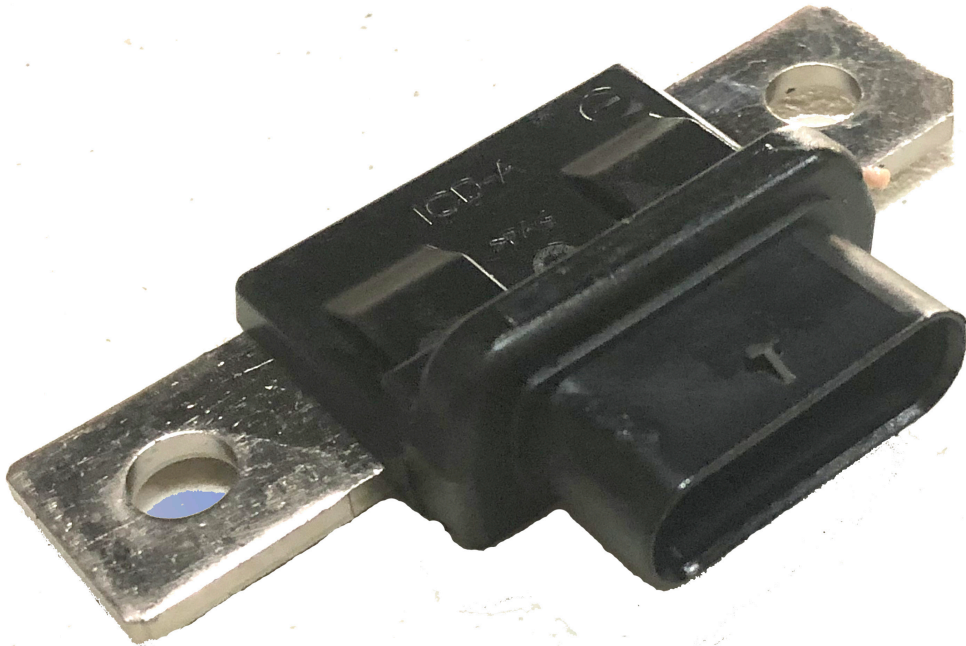


EVTV User Guide



Isabellenheutte
ICD-A-500-CAN1-12
CAN Current Sensor

INTRODUCTION

The Isabellenheutte ICD-A-500-CAN1-12 is a highly compact precision current measurement device. The system uses shunt-based current measurement technology for maximum accuracy and CAN 2.0A communications to provide current information remotely.

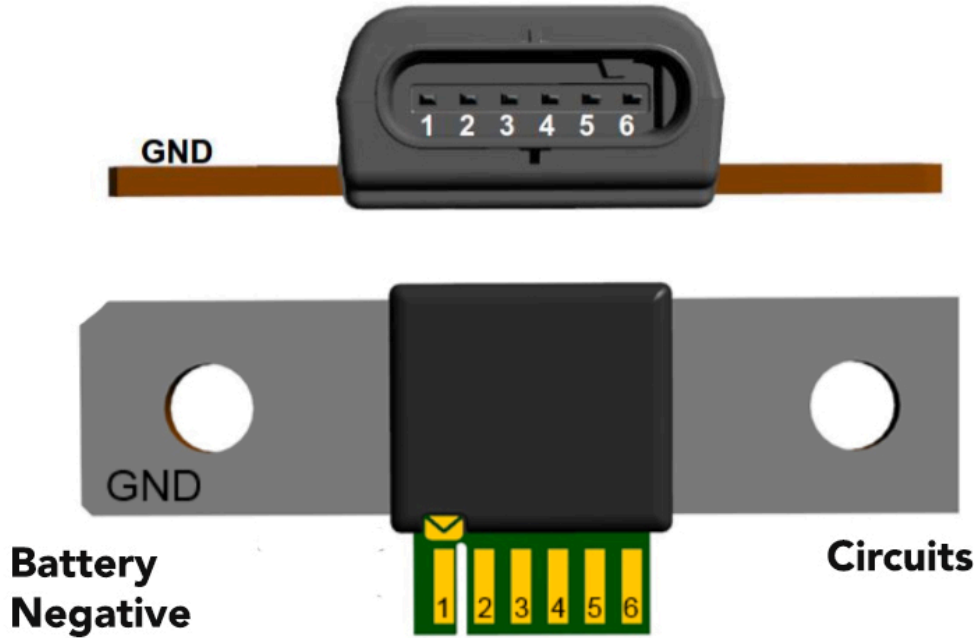
It consists of a 16Bit ADC for measurement acquisition and a microcontroller for processing and communication purposes. The current measurement value is available within a 24Bit range. An internal sampling rate of 1 kHz is used and a moving averaging filter can be programmed.

The communication is based on CANbus 2.0a/b with a data rate up to 1Mbit/s. A CANbus data base container (DBC) is available to support fast system integration. The ICD series is covered with a molded housing to resist a wide range of environmental influences. With the highly compact design it can easily be integrated where installation space is limited.

Full documentation for the device is available at:

https://www.isabellenhuette.de/fileadmin/Daten/Praezisionsmesstechnik/Datasheet_ICD_V1.00.pdf

SPECIFICATIONS



Mating: TE Connectivity 1-1718646-1
Contacts: 1452671-1

PIN	USE
1	CAN LOW
2	CAN HIGH
3	12v return
4	+12vdc
5	Not used
6	Not used

Operation conditions

Parameter	min.	typ.	Max.	Unit
Operating temperature	-40		+105	°C
Storage temperature	-40		+125	°C
Supply voltage	5.5	12	26	V
Current consumption	< 15	< 30	< 50	mA
Current consumption In sleep mode		< 100	< 250	µA
Re-/ Startup time		250		ms
Waiting time power on/off	2			ms

Maximum ratings

Parameter	Min	max	Unit
Storage temperature	-40	+125	°C
Supply voltage	-36	+38	V

Parameter	500
Extended load (max. time)	
5min	±730 A
30s	±860 A
10s	±1000 A
1s	±2700 A
200ms	±6000 A

Supply voltage measurement

Parameter	Min	Max	Unit
Measurement range	+5.5	+ 26	V
Initial accuracy		± 0.1	% of rdg
Total accuracy		± 0.8	% of rdg
Offset		± 35	mV
Noise		± 60	mV (rms)
Resolution		0.5	mV

Temperature measurement (on-chip)

Parameter	Min	Max	Unit
Measurement range	-40	+125	°C
Initial accuracy		± 3	°C
Total accuracy		± 5	°C
Resolution		0.1	°C

Communication

Interface	Specification	Speed	Max. number of Unit
CANbus	2.0 a/b	250kbit/s; 500kbits/s; 1Mbit/s	6

	Direction	MIN	MAX	UNIT	
V _{CC}	Supply voltage for CAN	4.75	5.25	V	
V _I or V _{IC}	Voltage at any bus terminal (separately or common mode)	-12	12	V	
V _{max}	Voltage at any bus terminal (max. rating)	-26	26	V	
V _{IH}	High-level input voltage	TXD,S	2	5.25	V
V _{IL}	Low-level input voltage	TXD,S	0	0.8	V
V _{ID}	Differential input voltage		-6	6	V
I _{OH}	High-level output current	Driver	-70		mA
		Receiver	2		mA
I _{OL}	Low-level output current	Driver	70		mA
		Receiver	2		mA

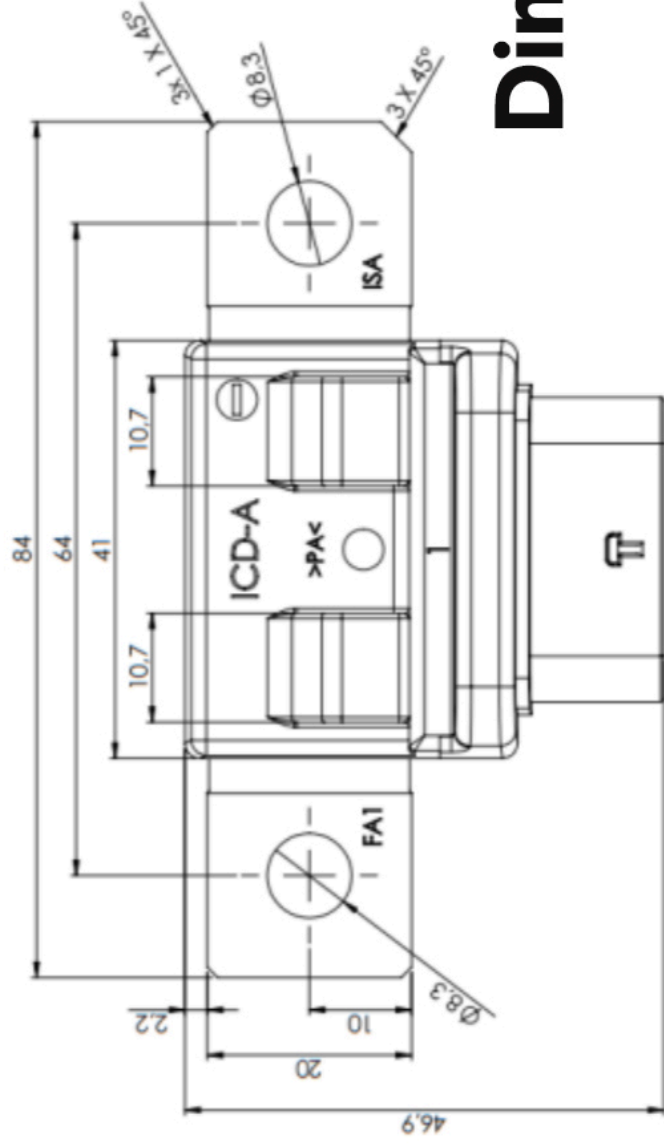
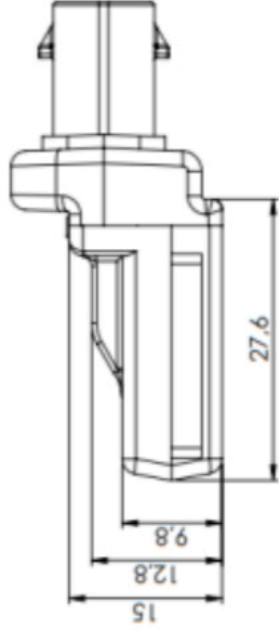
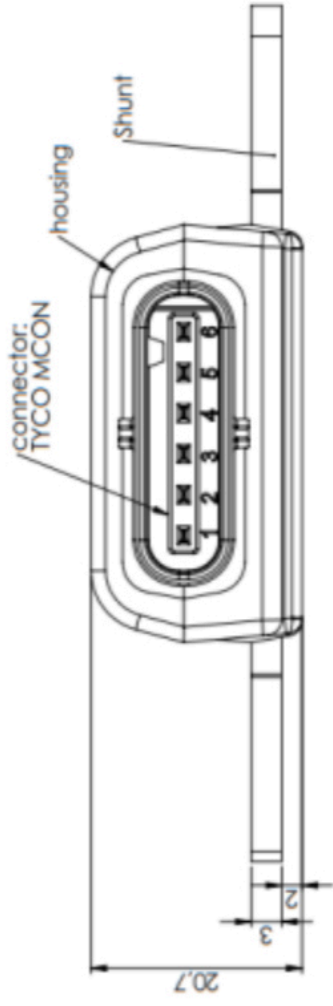
Current measurement

Parameter		Unit
Nominal measurement range (depending on shunt)	±500	A
Power loss	< 9	W
Overcurrent measurement range	±2500	A
Initial accuracy		% of rdg ¹
Total accuracy		% of rdg ¹
Offset	<± 60	mA
Noise ²	<± 35	mA (RMS)
Linearity ³		% of range
Resolution		mA

Default values

If using the Reset command with parameter 0x01 (reset all parameters to default) the following values are reset to its related default value.

Message	Parameter	Value
Operation Mode	Mode Output rate Current direction	0x01 - continues 0x01 - 1 ms 0x00 - positive
Result Ah Counter	Ah-Counter value	0x00 - 0 mAs
Averaging	Average value	0x01 - 1
OC limit	Activation limit positive Activation limit negative	0x00 00 00 0x00 00 00
CAN configuration	Baudrate CAN mode	0x01 - 500k Baud 0x00 - 2.0 a
CAN ID request	CAN ID	0x500
CAN ID response	CAN ID	0x501
CAN ID result	CAN ID	0x502
User PW	Mode 0x01	0x30; 0x30; 0x30; 0x30; 0x30; 0x30
Result MSG Structure	Result MSG Structure value	0x00 - UBat
Up time	Up time value	0x00 00 00 00
Status Bits	Measurement status bits value System status bits value	0x00 00 0x00 00
Event Counter	All measurement counter values All system counter values	0x00 00 0x00 00



Dimensions

ICDA LIBRARY

EVTV has developed an Arduino ESP32 Class Library for the ICD-A current sensor device. This allows you to easily use the current sensor in any Arduino style program for the ESP32.

The Class library comes in two files: ICDA.h and ICDA.cpp.

To use this in your program, add the following include lines to the first lines of your program:

```
#include "esp32_can.h" //http://github.com/collin80
#include "ICDA.h"
```

The sensor requires the ability to send and receive CAN messages and so the CAN library for the ESP32 also developed also by EVTV must be included.

To instantiate an object of Class ICDA:

```
ICDA Sensor;
```

This will create an ICDA object titled **Sensor**.

To initialize the **Sensor** object, you must initialize it by specifying a CAN port to use,

```
Sensor.begin(0); //Initialize an ICDA object using port  
CAN0
```

The ICD object will send CAN 0x502 messages with current measurement at a regular time period of 50ms by default. You can optionally specify the number of milliseconds between frame transmissions.

```
Sensor.begin(1,100); //Initialize ICDA object using  
port CAN1 using a rate of 100ms between each measurement  
0x502 frame.
```

Finally, the ICDA library can average the current values received using a 100 element round robin float variable array. Larger values have the effect of smoothing the readings but cause a measurable lag in changes in current. Smaller values make the readings more responsive, but also more erratic.

```
Sensor.begin(1,75,50); //Initialize ICDA object to use  
port CAN1 using a rate of 75ms between each measurement  
0x502 frame and using a 50 element averaging array filter.
```

The ICDA device will send current measurement data in CAN message ID **0x502**. To process this frame, you must receive it in your main program loop and forward it to the Sensor object as follows:

```
Sensor.getFrame(&inFrame1);
```

```
//Process in incoming 0x502 CAN frame named inFrame1 of data type
```

Example:

```
CAN_FRAME inFrame1;  
if (CAN0.available())  
  {  
    CAN0.read(inFrame1);  
    switch (inFrame1.id)  
      {  
        case 0x502:  
          Sensor.getFrame(&inFrame1);  
          break;  
      }  
  }
```

This sets up a CAN_FRAME data structure and checks the CAN0 port to see if anything is incoming. If so, it stores it in inFrame1. It then checks to see if inFrame1.id equals 0x502 and if so, forwards it to our Sensor object.

```
Sensor.deFAULT();
```

```
//Reset the ICD-A-500-CAN1-12 current sensor to default values. See ICD-A data sheet.
```

```
Sensor.resetAH();
```

Resets AH variable calculated **Sensor.AH** but also resets the onboard sensor amp hour counter **Sensor.ah**

There are also some ICDA public variables you can access and one of them is rather important.

```
Sensor.Voltage
```


The library will calculate kilowatts and kilowatt hours for you based on current measurements. But the ICDA device does not measure voltage. And so you should periodically set the **Sensor.Voltage** variable to your measured battery pack voltage.

For example, if you keep your battery pack voltage in the variable **packvoltage**:

```
Sensor.Voltage=packvoltage;
```

The ICDA class will also readily calculate State of Charge(SOC) but it needs your total pack size in ampere-hours.

```
Sensor.capacity=packsizeinamphours;
```

List of public variables available in IDCA.

```
float Amperes; // current in Amperes  
float AH; //accumulated ampere-hours calculated  
float ah; //accumulated ampere-hours by sensor  
float MaxNegAmps; //Peak discharge amps this session  
float MaxPosAmps; //Peak charge amps this session  
float Voltage; //Pack voltage, SET THIS for kw calcs  
float KW; //Instantaneous POWER using Voltage  
float KWH; //Accumulated KWH  
float SOC; //SOC – capacity+AH/capacity  
float capacity; //Pack capacity in AH  
int framecount; //Number of 502 frames received  
float chargingAH; //Total AH charged  
float dischargingAH; //Total AH discharged  
float chargingKWH; //KWH charged  
float dischargingKWH; //KWH discharged  
uint8_t arraysize; //Averaging filter – SET anytime
```