EPICS ADS Device Support

Software Architecture

|  |  |
| --- | --- |
| Revision: |  |
| Status: |  |
| Project: | I-GroundZero-ADS |
| Document ID: |  |
| File: |  |
| Owner: | Andrej Manojlović |
| Last modification: | September 7, 2020 |
| Created: | April 24, 2018 |

Glossary of Terms



|  |  |
| --- | --- |
| GUI | Graphical User Interface |
| HW | Hardware |
| SW | Software |
| ADS | Automation Device Specification |
| AMS | Automation Message Specification |
| PLC | Programmable Logic Controller |
| EPICS | Experimental Physics and Industrial Control System |
| IOC | Input/Output Controller |
| CA | Channel Access |
| PV | EPICS Process Variable |

Purpose and Scope

The purpose of this document is to describe the organization of the main software components of the EPICS ADS devices support software and its relationship with other software.

References

1. EPICS ADS project proposal and requirements description:  
   <https://git.cosylab.com/groundzero/g0-mngmnt/blob/master/proposals/CSL-G0-001-INT-aman-ADS-PRP.docx>
2. Beckhoff ADS introduction:  
   <https://infosys.beckhoff.com/content/1033/tcadscommon/html/tcadscommon_intro.htm?id=898081192215463875>
3. Beckhoff ADS client library:  
   <https://github.com/Beckhoff/ADS>
4. asynDriver:  
   <https://epics.anl.gov/modules/soft/asyn/>
5. Dynamic Port Driver:  
   <https://git.cosylab.com/sisaev/DynamicAsynPortDriver>
6. EPICS Modbus device support:  
   <http://cars9.uchicago.edu/software/epics/modbusDoc.html>
7. Beckhoff TwinCAT 2:  
   <https://www.beckhoff.com/english.asp?twincat/default.htm>
8. Beckhoff PLC data types:  
   <https://infosys.beckhoff.com/english.php?content=../content/1033/tcplccontrol/html/tcplcctrl_plc_data_types_overview.htm&id>
9. Beckhoff AMS TCP packet structure:  
   <https://infosys.beckhoff.com/content/1033/tc3_adscommon/html/tcadsamsspec_amstcppackage.htm?id=3991988313072350261>
10. AdsNotificationAttrib structure description: <https://github.com/Beckhoff/ADS/blob/7758220f831e295d45d4dcffc79c8bbfdde666f5/AdsLib/AdsDef.h#L352>

Table of Contents

[1. General Description 4](#_Toc45630006)

[1.1. ADS protocol features supported 4](#_Toc45630007)

[2. ADS device support architecture 6](#_Toc45630008)

[2.1. Constraints 6](#_Toc45630009)

[2.2. System overview 6](#_Toc45630010)

[3. System interactions 8](#_Toc45630011)

[3.1. Device support initialization 8](#_Toc45630012)

[3.2. Write to ADS device 8](#_Toc45630013)

[3.3. Read from ADS device 9](#_Toc45630014)

[3.3.1. Passive/periodic scanned records 9](#_Toc45630015)

[3.3.2. I/O event scanned records 9](#_Toc45630016)

[3.4. ADS device is disconnected 9](#_Toc45630017)

[3.5. ADS device is reconnected 10](#_Toc45630018)

[3.6. Reporting functionality 10](#_Toc45630019)

[4. Device support configuration 11](#_Toc45630020)

[4.1. Global configuration 11](#_Toc45630021)

[4.2. Configuring ADS connections 11](#_Toc45630022)

[4.3. Database record configuration 12](#_Toc45630023)

[4.3.1. Example record configuration 13](#_Toc45630024)

# General Description

The purpose of the software component EPICS ADS device support is to enable EPICS database records to read and write values to and from ADS variables and registers on ADS capable devices (Beckhoff PLCs), also called ADS servers. The ADS device support is composed of multiple software components described in section 2.2. The specific software component that is the focus of this document is referred as *ADS port driver*.

ADS [2] is a client/server interface or a set of commands (further referred as *ADS protocol*) that is used for communication with ADS capable devices. The basic set of ADS commands is:

* Read/write ADS data.
* Add/remove ADS data notification (subscribe to value changes on the device).
* Read ADS device state and information.

Note that the ADS protocol is encapsulated within the AMS protocol [9], which is itself usually encapsulated within the TCP protocol.

## ADS protocol features supported

The following ADS features are supported by the ADS device support:

* Read/write data from/to a register on ADS server.
* Read/write data from/to a variable on ADS server.

The ADS notification feature is no longer supported by ADS device support as of v2.0.0.

The term register is used to describe an address on the ADS service, which corresponds to a specific I/O terminal's input or output channel raw value. For example, the second (boolean type) input channel on the first 16-channel digital input terminal can have an address specification such as this: Port: 300, IGrp: 0x14002, IOffs: 0x349, Len: 1 [Figure 1].

The term variable or symbolic name is used to describe a named data block in the memory of the program running on the ADS server. Each variable has a specified size and type (e.g. INT, UINT, FLOAT, array of BOOL, etc.). An example variable name can be MAIN.input\_temperature, where the first part MAIN represents the name of the running program (in this case the PLC main program) and the second part input\_temperature the name of the variable used in that PLC program. Depending on the PLC program, variables can either be linked to physical I/O terminals (e.g. to a specific 0..10 V analog input channel connected to a temperature sensor), or they can be logical – derived from I/O channels (e.g. converted raw input value from temperature sensor to degrees Celsius), or representing some internal state (e.g. program running cycle counter).

Note that with ADS, both registers and variables are finally accessed in the same way: using the *port*, *group*, *offset* and *length* specifiers. The difference is that for variables, the symbolic name is first dereferenced into an ADS group and offset specifiers (further referenced in this document as handle).

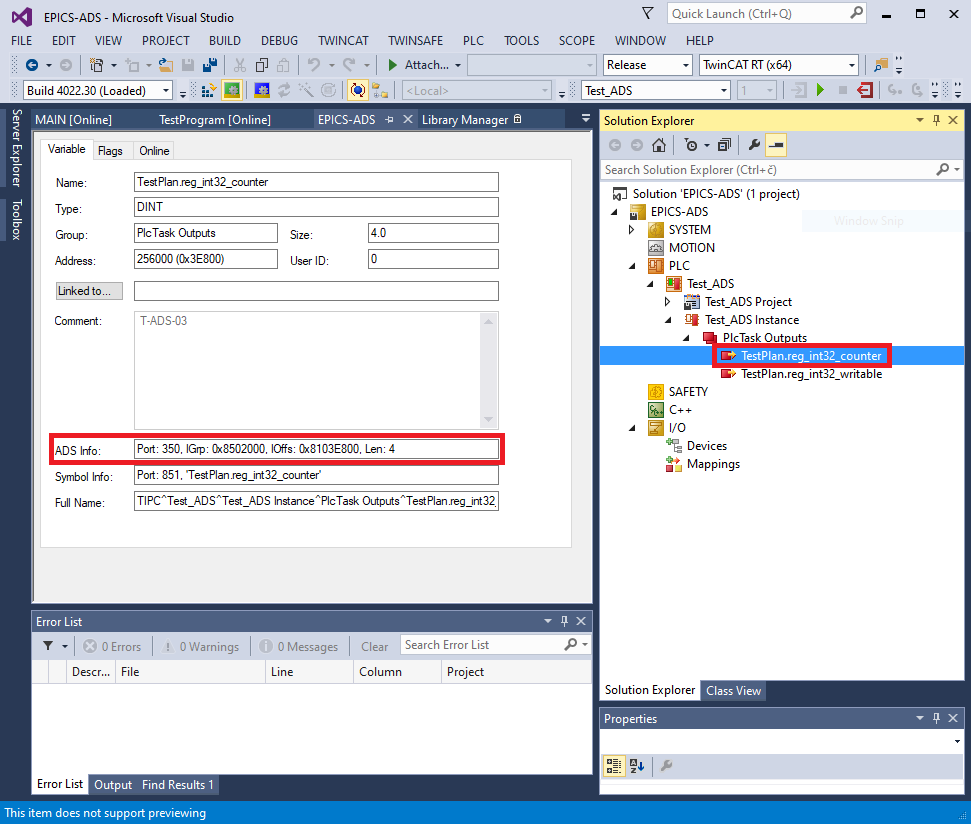


Figure 1: Example ADS address information of “TestPlan.reg\_int32\_counter” variable in the PLC program

# ADS device support architecture

## Constraints

The following constraints were taken into consideration during the development of the architecture:

* The ADS device support software must work on Windows and Linux operating systems.
* The IOC using the device support must be able to open ADS connections to multiple ADS servers.

## System overview

The major software components of the ADS device support are [Figure 2]:

* EPICS base: The EPICS IOC containing database records, which read/write to/from specific ADS variables or registers.
* asynDriver: General purpose library for interfacing device specific code to low level drivers. Extended by Dynamic port driver and ADS port driver.
* Dynamic port driver: Simplifies integration of devices with lots of I/O signals of different types, which are handled in a similar way. The ADS port driver uses (extends, to be more accurate) Dynamic port driver to keep track of which ADS variables correspond to which database record in the IOC.
* ADS port driver: Implements the layer between database records and Beckhoff ADS library. It handles the read/write requests initiated by database records with SCAN=Passive/periodic and converts them to ADS client library calls. Records with SCAN=I/O Intr have their values updated whenever the corresponding ADS variable changes its value.
* Beckhoff ADS client library: Implements the AMS/ADS protocol on top of TCP/IP.

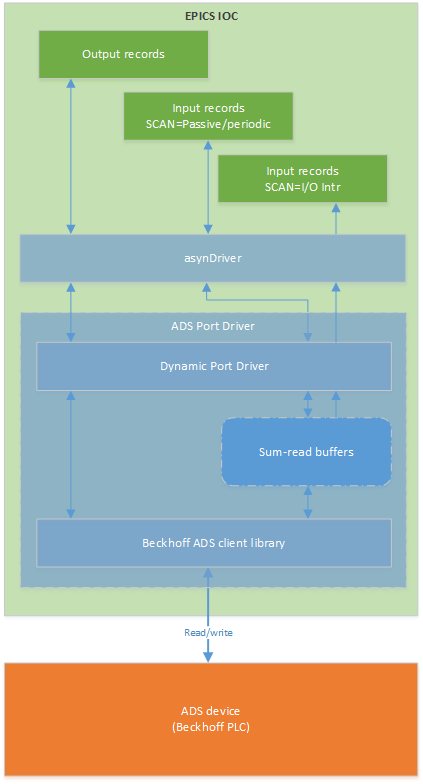


Figure 2: Software components and communication flow diagram

# System interactions

## Device support initialization

The device support per-port initialization is composed of the stages listed below. The local AMS address is assumed to have been configured beforehand, using AdsSetLocalAMSNetID described in section [4.1].

1. ADSPortOpen: The IOCSH command creates an asyn port, which is later referenced in records INP/OUT fields.
2. iocInit: Assigns an asyn *reason* index and stores information about the desired ADS operation for that record.
   * Each record registered for the port instance is passed to the *drvUserCreate* method.
   * The INP/OUT field is parsed to determine what ADS operation should be performed for the record. E.g. OUT=@asyn($(PORT), 0, 0) INT W P=PLC\_TC3 V=Main.variable implies a 16-bit integer write to an ADS variable called Main.variable. This information is then stored in a *PVInfo* object by the ADS device support.
   * An asyn reason index is assigned to the record.
   * The *PVInfo* object is stored in multiple collections, which allow lookup either by operation type (read or write) and by asyn *reason*. E.g. when the corresponding record is later processed, the PVInfo lookup is done by asyn *reason*.
   * When IOC initialization is done and *drvUserCreate* has been called for all corresponding records, the port driver allocates the buffers necessary to perform e.g. ADS sum-read operations.
3. Connect to ADS device: Opens a connection to the target ADS device.
   * The connect method is called automatically by asyn if the autoConnect parameter is set to TRUE.
   * A new ADS connection is established to the ADS device.
   * ADS variables (symbolic names) are dereferenced and handles are obtained.
   * The sum-read request buffers are initialized with variable handles.

If any of the steps above fails (e.g. the INP/OUT field syntax is incorrect), the error is logged and return *asynError* is returned.

## Write to ADS device

ADS write operation occurs when an output record (e.g. ao, longout) is processed.

1. Depending on the records type and DTYP field (e.g. asynInt32), the port driver calls the matching handler method (e.g. writeInt32).
2. Using the reason parameter as key, the handler method looks up ADS mapping information (*PVInfo* object) for the record that triggered the write operation.
3. ADS mapping information is used to synchronously perform the ADS write request.
4. If the ADS write operation completes successfully, the handler method writes the ADS data value to the record and returns *asynSuccess*, indicating a successfully completed operation.

If any of the steps above fails, the handler logs the error and returns *asynError*.

|  |  |  |
| --- | --- | --- |
|  | In current ADS port driver version, a large number of simultaneous write requests can saturate the ADS connection and cause the system to become unresponsive and cause records to time out. |  |

## Read from ADS device

The port driver handles reading values from ADS device in the same way for all input records, regardless of their SCAN field specification (passive, periodic or *I/O Intr*). ADS variables corresponding to input records are grouped and assigned to a corresponding sum-read request and data buffer. The port driver then periodically reads the variables using ADS sum-read functionality.

This implies that input record processing will never trigger a synchronous read call to the ADS device. Instead, input record values are essentially read from a cache.

### Passive/periodic scanned records

When a SCAN=Passive/periodic input record (e.g. ao, longin) is processed, the following steps are executed:

1. Depending on the records type and DTYP field (e.g. asynInt32), the port driver calls the matching handler method (e.g. writeInt32).
2. Using the reason parameter as key, the handler method looks up ADS mapping information (*PVInfo* object) for the record that triggered the write operation.
3. Handler method reads the value from the corresponding sum-read data buffer.
4. The handler returns *asynSuccess* if the sum-read data buffer is marked as valid and the ADS read operation for the particular variable indicates success.

### I/O event scanned records

Records with SCAN=I/O Intr are processed when the port driver detects that the corresponding ADS variable changed. Port driver compares the latest value in sum-read data buffer and compares it with the previous sum-read operation value.

If the new and old values don’t match, the registered callback is invoked and the record is processed. The callback invoked depends on the record type (e.g. ai, waveform) and DTYP field (e.g. asynInt32, asynInt32ArrayIn).

## ADS device is disconnected

When any method communicating with the ADS device detects a communication problem, the port driver will attempt to disconnect from the ADS device to prevent any faulty operation. In such cases the disconnect method attempts to:

1. De-initialize (but not de-allocate) sum read request and data buffers to prevent further reading from them.
2. Release open ADS variable handles.
3. Close the AMS connection.

If any of the steps above fails (e.g. can't release ADS variable handles), the handler logs the error and attempts to finish the remaining operations.

## ADS device is reconnected

Reconnecting to the ADS is done by the port driver communication thread, which periodically attempts to re-establish ADS connection. Reconnection follows the steps 3 from the initialization procedure [3.1].

## Reporting functionality

The device support reports the following information when dbior or asynReport is called, depending on the interest level specified:

* Interest level 0 (no report):
  + Output of asynPortDriver (parent class) report method.
* Interest level 1 (summary report):
  + Output of previous interest level.
  + Total number of registered PVs.
  + Total number of ADS register addresses.
  + Total number of ADS variable handles.
* Interest level 2 (detailed report):
  + Output of previous interest level.
  + Summary of sum-read request and data buffers allocated.
  + Details of every ADS register address and datatype.
  + Details of every ADS variable name, datatype and its handle.
* **Interest level 3** (more detailed report):
  + Output of previous interest level.
  + Details about all sum-read request and data buffers allocated.

# Device support configuration

## Global configuration

AMS protocol specifies that each communication end-point must be identifiable by a 6-byte long unique AMS Net ID (e.g. "172.16.17.10.1.1"). This ID is logical and has no relation to an IP address, although it is sometimes set to the same value as the end-points IP address, with an additional ".1.1" suffix.

The ADS device support IOC shell command that sets the optional local (IOC) AMS Net ID for communication with ADS devices is: AdsSetLocalAMSNetID(ams\_net\_id)

This command must be called before opening any ADS connections. If not, the ADS client library automatically assigns a local AMS Net ID.

## Configuring ADS connections

The ADS device support enables the IOC to open multiple connections to multiple ADS devices [Figure 3]. Such a connection corresponds to a port in asyn terminology. To open an ADS connection the following IOC shell command is used:

AdsOpen(port\_name, ip\_addr, ams\_net\_id, sum\_buffer\_nelem, ads\_timeout, no\_auto\_connect, priority)

When opening a new port in the IOC shell, the following per-connection parameters are available:

* port\_name: Asyn port name that uniquely identifies the ADS connection (e.g. "ADS-02").
* ip\_addr: Remote ADS device IP address (e.g. "10.5.1.100").
* ams\_net\_id: Remote ADS device AMS Net ID address (e.g. "10.5.1.100.1.1").
* ***sum\_buffer\_nelem*** (optional): The maximum number of PVs that sum-read request and data buffers can contain.
* ads\_timeout: Timeout value for internal ADS function calls (e.g. 5000 ms), corresponding to AdsSyncSetTimeoutEx function [3].
* no\_auto\_connect (optional): Disables auto-connect functionality for the port instance if set to a non-zero value.
* priority (optional): Thread priority for the port instance. If not specified or is set to zero, epicsThreadPriority is used. See epicsThread.h found in EPICS base for thread priority values.

Due to ADS protocol constraints, it is not possible to open multiple connections to a single ADS device from the same client. More precisely, the ADS server will accept only one connection from different ADS clients that have the same AMS Net ID, or from multiple ADS clients with different AMS Net IDs but the same IP address.

|  |  |  |
| --- | --- | --- |
|  | The ADS device will close the existing connection if another client connects from the same IP address and AMS ID. |  |

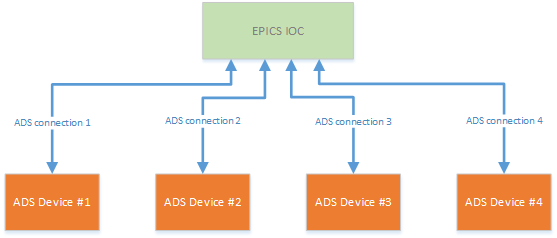


Figure 3: EPICS IOC with multiple connections to ADS devices

## Database record configuration

Database records are configured as follows:

* DTYP field specifies the asyn interface (e.g. *asynInt32*, *asynFloat64*, *asynInt8ArrayIn*) and is set according to the record type. Some records allow different asyn interfaces to be set. For more information see asynDriver distribution's dbd directory, devAsyn\*.dbd (e.g. devAsynInt32.dbd) files.
* INP and OUT fields uses the asyn drvParams optional parameter (see [4], section Generic Device Support for EPICS records) to specify the ADS register/variable that the record is connecting to, number of elements and the operation to be performed (read or write). For example: INP=@asyn(plc-01 0 0) USINT R P=300 G=0x14002 O=0x349 or OUT=@asyn(plc-02 0 0) REAL[10] W P=PLC\_TC3 V=Main.SetCalibration.
* SCAN field determines when the record is processed and, depending on the record type, how their value is handled:
  + Output records with SCAN=Passive/periodic write to their corresponding ADS variable immediately when they are processed.
  + Input records with SCAN=Passive/periodic have their VAL field populated with the latest value from their ADS variable's corresponding sum-read data buffers when processed.
  + Input records with SCAN=I/O Intr have their VAL field populated immediately after the port driver detects their corresponding ADS variable's value in the sum-read databa buffer has changed.

The format used to specify the ADS address/variable in the INP/OUT fields is: DATA\_TYPE[NELEM] OPERATION TARGET

Where:

* ***DATA\_TYPE*** part specifies one of the supported PLC data types, e.g. USINT, LREAL, STRING, etc.
* ***NELEM*** (optional) is used to specify number of elements for array access, as well as to specify the length of STRING PLC variables. The default value for the parameter is 1.
* ***OPERATION***specifies if the PLC variable is read (R) or written (W).
* ***TARGET*** specifies the ADS register or variable:
  + For register access:
    - ***P***= ADS port in string or numerical format, e.g. P=IO, P=300 or P=0x12c.
    - ***G***= ADS group, e.g. G=0x14002 or G=81922.
    - ***O***= ADS offset, e.g. O=0x349 or O=841.
  + For variable access:
    - ***P***: ADS port in string or numerical format. The same parameter constraints apply as for register access.
    - ***V***: ADS variable name in string format, e.g. V=Main.temperature.

### Example record configuration

record(ai, "read\_real\_register") {

field(DESC, "Read updates from REAL register on ADS device")

field(SCAN, "I/O Intr")

field(DTYP, "asynFloat64")

field(INP, "@asyn($(PORT), 0, 0) REAL R P=IO G=0x14002 O=0x349")

}

record(longout, "write\_variable") {

field(DESC, "Write to UINT variable on ADS device")

field(SCAN, "Passive")

field(DTYP, "asynInt32")

field(OUT, "@asyn($(PORT), 0, 0) UINT W P=851 V=Main.some\_setpoint")

}

record(waveform, "read\_real\_array") {

field(DESC, "Read from REAL array on ADS device")

field(SCAN, "Passive")

field(FTVL, "FLOAT")

field(NELM, "24")

field(DTYP, "asynFloat32ArrayIn")

field(INP, "@asyn($(PORT), 0, 0) REAL[24] R P=PLC\_TC3 V=Main.arr\_reals")

}

record(waveform, "write\_byte\_array") {

field(DESC, "Write to BYTE array on ADS device")

field(SCAN, "Passive")

field(FTLV, "UCHAR")

field(NELM, "100")

field(DTYP, "asynInt32ArrayOut")

field(INP, "@asyn($(PORT), 0, 0) BYTE[100] W P=PLC\_TC3 V=Main.arr\_bytes")

}