Digital Wave Height Gauge
Command Protocol

Issue: 0.6
Status: Draft
08-July-2014

Prepared by
Akamina Technologies Inc.

Table of Contents

1 OVERVIEW

The original AWP-300 was designed with CAN Bus and RS-485 physical interfaces. Both feature high bit rates, long cable lengths and can be deployed in a multi-dropped configuration. This configuration allows many AWP-300s to be connected to the same network.

A messaging protocol has been developed to allow a network master to manage and collect data from multiple devices. This message protocol is a modified Modbus protocol. The modifications were made to allow the data field of a packet to fit into 8 bytes or less.

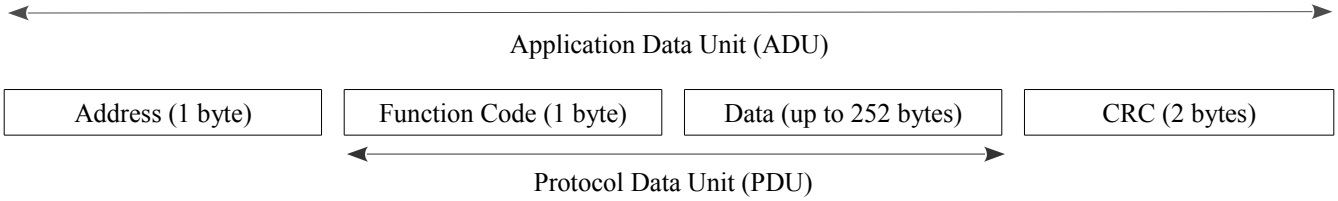
The Modbus protocol specification can be found at www.Modbus-IDA.org

The current messaging implementation is only supported on the CAN Bus interface.

2 COMMAND PROTOCOL

2.1 Modbus Command Packet

The general Modbus frame format is shown in the figure below.



A Modbus packet can be up to 256 bytes but we will restrict the size considerably by limiting the PDU portion of the packet to 8 bytes from 253 bytes.

The basic PDU consists of a 1-byte function code followed by up to 7 bytes of data. The data portion is limited to 7-bytes so that the function code and data will fit into a 64-bit CAN Bus data field.

The wave height gauge (device) is designed such that a single host (master) can manage a number of gauges connected on a multi-drop network. Each device is given a unique address as well as a multicast address. To communicate with a device, the master will prepare a request packet that is sent to one or all devices on the network. The devices that are addressed will then prepare a single response packet that is sent back to the master. In general, a device will not send a packet to the master unless it receives a request packet. The exception is when the device is in a automatic-upload mode where acquired samples are automatically sent to the master at the prescribed sampling frequency.

Devices always listen to the multicast address of 0x1F. Additionally, the master can set the Device ID of a device to a value between 0x01 and 0x1E. This limits the maximum number of devices on a multi-drop bus to 30 since both the address 0x00 and the address 0x1F can not be used as unique Device IDs. Once the device has been assigned a Device ID, it will listen for messages addressed to the multicast address as well as the Device ID. The master address is 0x00. All response packets will have the address field of the frame set to 0x00.

A subset of the standard Modbus function codes and a number of user-defined function codes will be used to manage the wave height gauge (device). Management typically involves reading or writing one or more registers by sending one or more request frames to the devices on the network. Standard registers are 2 bytes but a small number of special-purpose registers are defined as 4 bytes. These 4-byte registers are only accessible using user-defined function codes.

Modbus uses the big endian representation so that 0x1234 would be sent 0x12 followed by 0x34.

2.2 Supported Function Codes

The function codes that are supported within the digital wave height gauge family are shown in the table below.

Function Code (hex)	Description
0x04	Read Input Registers – read one or more input registers. Technically this function code is to be used to read read-only registers but it is used to read both read-only and read/write registers. The 8-byte limitation imposed by CANbus limits the number of registers that can be read with a single message to 3
0x06	Write Single Register – write one register
0x10	Write Multiple Registers – write one or more registers. The 8-byte limitation limits the number of registers that can be written with a single message to 2.
0x42	Read Single 32-bit Register
0x43	Write Single 32-bit Register

Table 1: Supported Function Codes

2.3 Supported Register Types

Register Ids and register values are both 16-bit values. There are basic register types that are defined are shown in the following table.

Type	Valid Function Codes	Description	Expected Use	ID Range
Action Registers	0x06	Write-only registers that cause a specific action to occur. These are not registers in the typical sense as no value is stored in the register. A write access to the register caused the defined action to be taken	Common – for general management of the device	0x8000 to 0x80FF
Input Registers	0x04	Read-only registers	Common	0x4000 to 0x40FF
Holding Registers	0x04, 0x06, 0x10, 0x42, 0x43	Read-write registers	Common	0xC000 to 0xC0FF

Table 2: Supported Register Types

The tables of defined register Ids for each of these register types are shown below.

Data	Type	ID	Description
Execute Config	Action	0x8000	Force the sensor to compute the optimal internal capacitor settings for the water level
Clear Config	Action	0x8001	Clear the internal capacitor settings
Adjust Min / Max	Action	0x8002	Optimize the sensor output range. For analogue output type sensors, this will force the sensor to output voltages in a +/- 4.5 V range; for digital output type sensors, this will force the sensor to output values in the range +/- 29,500. Note that the Execute Config action must have been completed before the min/max algorithm can be run. An error code of 6 (parameter can not be set now) will be returned if the config command has not been executed first.
Reset Sensor	Action	0x8004	Reset the sensor processor
Reset Sample Error Count	Action	0x8005	Reset the count of the number of samples acquired where an error occurred in the acquisition
Prepare For Calibration	Action	0x8006	Reset any internal registers in preparation for a calibration.
Factory Reset	Action	0x8007	Return internal register values to default values

Table 3: Common Action Registers (Write-only)

Data	Type	ID	Description
Serial Number	Input	0x4000	16-bit unsigned serial number assigned to the sensor during manufacturing
Date of Manufacture	Input	0x4001	32-bit time_t indicating the number of seconds since the Unix epoch of Jan 01, 1970
Bootloader Version	Input	0x4002	16-bit unsigned bootloader version number
Application Version	Input	0x4003	16-bit unsigned application version number
Data Sample	Input	0x4004	16-bit integer value. The most recent data value acquired
Data Status	Input	0x4005	16-bit unsigned data sample status
Device ID	Input	0x4006	16-bit unsigned device ID. See 2.3.1 Special Cases for more information.
Sample Error Count	Input	0x4007	16-bit unsigned count of the number of samples acquired where an error occurred in the acquisition

Table 4: Common Input Registers (Read-only)

Data	Type	ID	Description
Serial Number	Holding	0xC000	16-bit unsigned serial number assigned to the sensor during manufacturing. See 2.3.1 Special Cases for more information.
Device ID	Holding	0xC001	16-bit unsigned device ID. The device can be addressed using this device ID or the multicast device ID. See 2.3.1 Special Cases for more information.
Immersion Depth	Holding	0xC002	16-bit unsigned value indicating the percentage of the wave probe length that is immersed in water when the config action is executed. This allows the electronics to be configured without the probe head being fully immersed. The value can be between 10 and 100.
Coaxial Cable Length	Holding	0xC00B	Length (in cm) of the coaxial cable connecting the probe head to the electronics. This value is used in the configuration of the electronics to determine the minimum and maximum capacitance of the probe head.

Table 5: Common Holding Registers (Read / Write)

2.3.1 Special Cases

2.3.1.1 Device ID

Device ID appears as both a read-only input register and a read/write holding register. It appears as a read-only input register after the data sample register and the data sample status register. This allows a read input registers request packet to request all 3 of sample, status and device id in a single response packet. This is essential in cases where the request packet is sent to the broadcast address and more than one device returns a reading. The master that requested the samples will be able to determine the device from which the sample was sent using the Device ID.

The Device ID also appears as a holding register so that it can be set to the desired value.

A read of the Device ID input register and a read from the Device ID holding register will return the same value.

2.3.1.2 Serial Number

Serial Number appears as an input read-only register, as a holding read/write register and as a write-once locked holding register. Each of these uses are described below.

2.3.1.2.1 Read-only Serial Number Register

This is the typical read-only mechanism to read the serial number.

2.3.1.2.2 Read/Write Serial Number Register

The read/write holding register form is used with the Write Multiple Registers function code to write both the Serial Number register and the Device ID register. This is the only mechanism that is supported to set the Device ID. The Serial Number value in the request packet must match the device serial number for the Device ID value in the request packet to be written into the Device ID register of the device. This is an extra layer of protection to be sure that the Device ID is correctly set on only one device in a multi-drop network.

2.4 Exceptions

An exception occurs when the device is unable to process the request packet. When an exception occurs, the response packet format is:

- the 8-bit address field set to 0x00 indicating that the packet is directed to the master
- the most significant bit of the 8-bit function code is set to 1 to indicate an exception. The remaining 7-bits are set to the function code sent in the request packet
- an 8-bit exception code is returned in the first byte of the data field

2.4.1 Exception Codes

The exception codes that can be returned by the device are shown in the table below. User-defined exception codes start at 0x40.

Code	Description
0x01	Illegal function – the request packet contained an unsupported function code
0x02	Illegal data address – one or more of the register addressed in the request packet is unsupported
0x03	Illegal data value – a value contained in a request field is not an allowable value. This is not used when the value to be written into a register is out of range – this exception is defined below. An example of a request that would result in an Illegal Data Value response is a request to read or write more registers than can be accommodated in a single request or response packet.
0x04	Slave Device Failure – an unrecoverable error occurred while the device was attempting to perform the requested action
0x05	Acknowledge – used only in conjunction with firmware upgrades. This feature is not yet supported
0x06	Slave Device Busy – also used only for firmware upgrades.
0x08	Memory Parity Error – also used only for firmware upgrades.
0x40	Request Not Yet Supported – the request packet is valid but the operation is not yet supported by the device
0x41	Read-Only Register – an attempt to write to a read only register
0x42	Write-Only Register – an attempt to read from a write-only register
0x43	Range Error – the value specified in a write operation is out of the range supported
0x44	Not Now – the request is not allowed at the current time. This typically indicates that a necessary condition required to handle the request has not been satisfied
0x45	Invalid device ID – the value specified in a device ID write operation is invalid since the serial number match failed or was not part of the write request

2.5 Packet Formats

The packet formats for the function codes that are supported are defined below. Both the request form and response form of the packets are defined. Only the format for the function codes that are supported in the current version of firmware on the device are defined.

Forcing the length of the PDU to 8 bytes or less required that some standard packet formats be modified. The most significant change is any packets that use the number of registers in a multi-register read or write. The standard calls for this to be a 2 byte field but it has been reduced to a single byte for this implementation. This

allows a 2 register write and a 3 register read to be fit into 8 bytes – both of which are required to manage the devices properly. Any other changes to the standard packet formats are noted below.

2.5.1 Read Input Registers (0x04)

```
/**
 * \brief Define the Read Input Registers request message structure.
 *  - funcCode - 8-bit function code. Set to 0x04 for this request type
 *  - adrHi    - high byte of the 16-bit starting address register number
 *  - adrLow   - low byte of the 16-bit starting address register number
 *  - numReg   - number of consecutive registers to be read
 */
typedef struct {
  uint8_t funcCode;
  uint8_t adrHi;
  uint8_t adrLow;
  uint8_t numReg;
} rInputRegReq_t;
/**
 * \brief Define the Read Input Registers response message structure.
 *  - funcCode  - 8-bit function code. Set to 0x04 for this response type
 *  - byteCount - number of bytes of data return 2*number or registers. Given
 *                the limitation on packet size, this field can be 2, 4 or 6
 *  - iReg0Hi   - high byte of the first 16-bit register requested
 *  - iReg0Low  - low byte of the first 16-bit register requested
 *  - iReg1Hi   - high byte of the second 16-bit register requested
 *  - iReg1Low  - low byte of the second 16-bit register requested
 *  - iReg2Hi   - high byte of the third 16-bit register requested
 *  - iReg2Low  - low byte of the third 16-bit register requested
 */
typedef struct {
  uint8_t funcCode;
  uint8_t byteCount;
  uint8_t iReg0Hi;
  uint8_t iReg0Low;
  uint8_t iReg1Hi;
  uint8_t iReg1Low;
  uint8_t iReg2Hi;
  uint8_t iReg2Low;
} rInputRegResp_t;
```

2.5.1.1 Packet format changes

- numReg field in the request packet is 1 byte rather than 2

2.5.2 Write Single Register (0x06)

```
/**
 * \brief Define the Write Single Register request message structure.
 *  - funcCode - 8-bit function code. Set to 0x06 for this request type
 *  - adrHi    - high byte of the 16-bit register number
 *  - adrLow   - low byte of the 16-bit register number
 *  - valHi    - high byte of the 16-bit register value
 *  - valLow   - low byte of the 16-bit register value
 */
typedef struct {
  uint8_t funcCode;
  uint8_t adrHi;
  uint8_t adrLow;
  uint8_t valHi;
```

```
  uint8_t valLow;
} wSingleRegReq_t;
/**
 * \brief Define the Write Single Register response message structure.
 *  - funcCode - 8-bit function code. Set to 0x06 for this response type
 *  - adrHi    - high byte of the 16-bit register number
 *  - adrLow   - low byte of the 16-bit register number
 *  - valHi    - high byte of the 16-bit register value
 *  - valLow   - low byte of the 16-bit register value
 */
typedef struct {
  uint8_t funcCode;
  uint8_t adrHi;
  uint8_t adrLow;
  uint8_t valHi;
  uint8_t valLow;
} wSingleRegResp_t;
```

2.5.2.1 *Packet format changes*

- none

2.5.3 Write Multiple Registers (0x10)

```
/**
 * \brief Define the Write Multiple Registers request message structure.
 *  - funcCode - 8-bit function code. Set to 0x10 for this request type
 *  - adrHi    - high byte of the 16-bit register number
 *  - adrLow   - low byte of the 16-bit register number
 *  - numReg   - number of consecutive registers to be written
 *  - val0Hi   - high byte of the first 16-bit register value
 *  - val0Low  - low byte of the first 16-bit register value
 *  - val1Hi   - high byte of the first 16-bit register value
 *  - val1Low  - low byte of the first 16-bit register value
 */
typedef struct {
  uint8_t funcCode;
  uint8_t adrHi;
  uint8_t adrLow;
  uint8_t numReg;
  uint8_t val0Hi;
  uint8_t val0Low;
  uint8_t val1Hi;
  uint8_t val1Low;
} wMultipleRegReq_t;
/**
 * \brief Define the Write Multiple Registers response message structure.
 *  - funcCode - 8-bit function code. Set to 0x10 for this response type
 *  - adrHi    - high byte of the 16-bit register number
 *  - adrLow   - low byte of the 16-bit register number
 *  - numReg   - number of consecutive registers to written
 */
typedef struct {
  uint8_t funcCode;
  uint8_t adrHi;
  uint8_t adrLow;
  uint8_t numReg;
} wMultipleRegResp_t;
```

2.5.3.1 Packet format changes

- numReg field in the request and response packets is 1 byte rather than 2
- standard includes a 1 byte byte count field in the request packet that has been removed; byte count is always twice the number of registers

2.5.4 Read Single 32-bit Register (0x42)

```
/**
 * \brief Define the Read Single 32-bit Input Register request message structure.
 *  - funcCode - 8-bit function code. Set to 0x42 for this request type
 *  - adrHi    - high byte of the 16-bit starting address register number
 *  - adrLow   - low byte of the 16-bit starting address register number
 *  - numReg   - number of consecutive registers to be read this must be 1
 */
typedef struct {
  uint8_t funcCode;
  uint8_t adrHi;
  uint8_t adrLow;
  uint8_t numReg;
} rInput32RegReq_t;
/**
 * \brief Define the Read Single 32-bit Input Register response message structure.
 *  - funcCode    - 8-bit function code. Set to 0x42 for this response type
 *  - byteCount   - number of bytes of data returned =  4*number or registers.
 *                  Given the limitation on packet size, this field can
 *                  only be 4
 *  - iRegHighest - highest byte (3) of the 32-bit register requested
 *  - iRegHigh    - high byte (2) of the 32-bit register requested
 *  - iRegLow     - low byte (1) of the 32-bit register requested
 *  - iRegLowest  - lowest byte (0) of the 32-bit register requested
 */
typedef struct {
  uint8_t funcCode;
  uint8_t byteCount;
  uint8_t iRegHighest;
  uint8_t iRegHigh;
  uint8_t iRegLow;
  uint8_t iRegLowest;
} rInput32RegResp_t;
```

2.5.4.1 Packet format changes

- this is a user-defined format so there is no standard that applies

2.5.5 Write Single 32-bit Register (0x43)

```
/**
 * \brief Define the Write Single 32-bit Register request message structure.
 *  - funcCode   - 8-bit function code. Set to 0x43 for this request type
 *  - adrHi      - high byte of the 32-bit register number
 *  - adrLow     - low byte of the 32-bit register number
 *  - valHighest - highest byte (3) of the 32-bit register value
 *  - valHigh    - high byte (2) of the 32-bit register value
 *  - valLow     - low byte (1) of the 32-bit register value
 *  - valLowest  - lowest byte (0) of the 32-bit register value
 */
typedef struct {
  uint8_t funcCode;
```

```
  uint8_t adrHi;
  uint8_t adrLow;
  uint8_t valHighest;
  uint8_t valHigh;
  uint8_t valLow;
  uint8_t valLowest;
} wSingle32RegReq_t;
/**
 * \brief Define the Write Single 32-bit Register response message structure.
 *  - funcCode   - 8-bit function code. Set to 0x43 for this response type
 *  - adrHi      - high byte of the 32-bit register number
 *  - adrLow     - low byte of the 32-bit register number
 *  - valHighest - highest byte (3) of the 32-bit register value
 *  - valHigh    - high byte (2) of the 32-bit register value
 *  - valLow     - low byte (1) of the 32-bit register value
 *  - valLowest  - lowest byte (0) of the 32-bit register value
 */
typedef struct {
  uint8_t funcCode;
  uint8_t adrHi;
  uint8_t adrLow;
  uint8_t valHighest;
  uint8_t valHigh;
  uint8_t valLow;
  uint8_t valLowest;
} wSingle32RegResp_t;
```

2.5.5.1 *Packet format changes*

- this is a user-defined format so there is no standard that applies

2.5.6 Error Response

```
/**
 * \brief Define the Error response message structure.
 *  - funcCode - 8-bit function code. MSB set to indicate that the response is
 *               an exception
 *  - adrHi    - high byte of the 16-bit register number
 *  - adrLow   - low byte of the 16-bit register number
 *  - valHi    - high byte of the 16-bit register value
 *  - valLow   - low byte of the 16-bit register value
 */
typedef struct {
  uint8_t funcCode;
  uint8_t errCode;
} errorResp_t;
```

2.5.6.1 *Packet format changes*

- none

3 CAN BUS IMPLEMENTATION

The initial implementation of a digital messaging protocol between a data collection device (master) and the wave height gauges (device) uses a CAN Bus. Some highlights of the CAN Bus implementation used with the AWP-300 are shown below:

- The CAN Bus protocol is designed to support a multi-master configuration. However, the messaging protocol implemented on the CAN Bus is designed for a single bus master (the data collection device / master) and multiple bus slaves (the wave height gauges).
- The CAN Bus base frame format of the data frame is used for most messaging. The base frame includes an 11-bit message identifier (Message ID) and up to a 64-bit data field. The base frame is 44 bits long plus the number of bits in the data field.
- The CAN Bus base frame remote frame type is used for requesting a sample from all devices on a bus. The remote frame is always 44 bits long and includes an 11-bit message identifier with no data field.
- A standard CAN Bus message does not include an address field. In the AWP-300 implementation, the Message ID field is used as an address field. The11-bit Message ID is used as follows:
    - bits 0 through 4 are used to identify the target of the message and bits 5 through 10 are used to identify the source of the message
    - a Message ID with the lower 5 bits set to 0x00 indicate that the message is addressed to the collection master
    - a Message ID with the lower 5 bits set to 0x1F indicate that the message is a multicast message intended for all devices on the bus (all AWP-300s)
    - Message IDs from 0x01 to 0x1E are used as unique addresses for up to 30 devices. The unique address for a device is its Device ID
    - even if a unique Device ID has been assigned, a device always listens to the multicast address as well
    - the Message ID set in a response packet by a device depends on whether the device has been assigned a unique Device ID or not. If the device has a unique Device ID, bits 0 through 4 are set to 0x00, bits 5 through 9 are set to its Device ID and bit 10 is set to 0. If the device has not been assigned a unique Device ID, bits 0 through 4 are set to 0x00 and bits 5 through 10 are set to the lower 6 bits of the serial number assigned to the device at the factory.
- the device to which a command is addressed will generate a single response packet for each command packet received
- the (up to) 8-byte data portion of the CAN message is mapped to the Protocol Data Unit (PDU) of the Modbus packet
- the length field of the CAN message is set to the number of bytes sent in the data portion of the CAN message

The CAN Bus protocol relies on the message identifier field being unique in order to avoid collisions on the bus. With the lower 5 bits of the Message ID for messages addressed to the data collection master set to 0, the upper 6 bits are used to provide a unique Message ID. There are two scenarios in which it would be possible to create a non-unique Message ID. One is where a device has been assigned a Device ID that is unique on one CAN Bus is moved to another CAN Bus where its Device ID is no longer unique. The second is where two or more devices

have not been assigned unique Device IDs and the lower 6 bits of their serial numbers are the same. In these cases, collisions will occur. Collisions will be detected by a CRC failure but higher level software will need to determine that Device IDs are not unique and take the appropriate action.

The CAN Bus is is shared by all of the nodes on the network. For the AWP-300, this includes the master as well as the wave height gauges. To receive a data sample from all devices on the network, the master sends a 44-bit remote frame with the Message ID set to 0x1F – the multicast address to which all devices will respond. Each device prepares a 108-bit data frame response that includes the sample data. Using the default bus speed of 500 kbps, the remote frame will occupy the bus for approximately 0.1 ms and the response message from each device will occupy the bus for approximately 0.22 ms. Allowing some additional time to decrease bus utilization, the maximum scan frequency in scans per second is approximately:

$$Fscan = \frac{1000}{0.2 + 0.4\mathrm{N}} \quad ,$$

where N is the number of AWP-300 devices on the bus. If N = 30, the practical maximum scan rate is 82 scans per second and if N = 5, the maximum scan rate is 450 scans per second. Note that the internal update rate of the device remains 200 updates per second so requesting samples from the AWP-300 faster than 200 scans per second will result in duplicate sample values.

The AWP-300 firmware is programmed to respond immediately to all data frame messages and remote frame messages sent to a unique Device ID. Messages sent in response to a remote frame message sent with the multicast address are delayed. The delay is equal to:

$$0.35 * (Device\ ID - 1)$$

in ms. This spreads out the responses so that all devices are not trying to access the bus at the same time. Given this delay arrangement, it is better to assign low Device IDs (i.e. if using 2 devices, use IDs 1 and 2 rather than 29 and 30).

For the CAN Bus rate of 500 kbps, the maximum bus length is 100 m.

3.1 Managing AWP-300s on a CAN Bus

In order to effectively manage a number of AWP-300s connected to a CAN Bus, the master should do the following:

1. Broadcast the request to read the serial number (Read Input Register; Serial Number) to all devices on the bus. Each AWP-300 will respond to the command by sending its serial number to the master.
2. Assign each AWP-300 a unique Device ID by broadcasting a Device ID set command for each device (Write Multiple Registers: Serial Number, Device ID). All devices will see each set command but the serial number will only match for one device. The error messages sent by the other devices are ignored.
3. Broadcast the Clear Config command (Write Single Register; Clear Config) to all devices on the bus. A specific device can also be targeted by setting the Message ID to the unique Device ID rather than the multicast address.
4. Broadcast the Execute Config command (Write Single Register; Execute Config) to all devices on the bus. A specific device can also be targeted by setting the Message ID to the unique Device ID rather than the multicast address.

To collect samples from the AWP-300s, the master sends a remote frame with the Message ID set to the multicast address. The master then processes the response data frames sent by the devices.

Other function codes and register IDs can be used as required for specific functionality.