STW Technic, LP

Mobile Controllers and Measurement Technology

Application Note

AN2002: CANOpen

Topics

- CANopen Concepts and Standards
- Implementing CANopen
- Troubleshooting

1 Introduction

CANopen is a communications protocol for embedded systems in automation.

2 CANopen Concepts and Standards

This section provides brief descriptions of CANopen concepts and standards, which are discussed in individual sections, which are as follows:

- Identifiers and Objects
- The Object Dictionary
- Accessing the Object Dictionary
- Handling Process Data
- Network Management

2.1 Identifiers and Objects

2.1.1 Identifiers

There are three different “identifiers” found in CANopen:

- Node ID
- Object Dictionary Indexes
- COB ID or CAN ID

The **Node ID** identifies a specific CANopen node. The range for allowed Node IDs is from 1 to 27.

The **Object Dictionary Indexes** are used to identify a specific variable within a node.

Pioneering New Technologies

The **COB ID or CAN ID** identifies a specific message on a network. This identifier is unique and is used for a specific communication channel, and can also include control bits (i.e. a bit that is used for enable/disable control).

2.1.2 Identifiers

There are four different "objects" found in CANopen

- The Object Dictionary (OD)
- The Process Data Objects (PDO)
- The Service Data Objects (SDO)
- The Connection Objects (COB)

The **Object Dictionary** is used to store variables and constants.

Process Data Objects are messages that contain process data.

Service Data Objects are messages that contain service and/or configuration data.

Connection Objects are used whenever a message needs to be assigned to implement a service.

2.2 The Object Dictionary

The Object Dictionary is like a table that stores all the data that is accessible from the network. Each CANopen node must implement its own Object Dictionary.

The Object Dictionary contains descriptions of CANopen configurations and the functionality of the node that it is stored in, and can be read or written to by other nodes. In addition, the Object Dictionary is used for other information that is used by the node. This information can also be used by other nodes on the network.

Object Dictionary Examples
Index 2000h stores a single 8-bit value: Index 2000h, Subindex 00h = 8-bit value **Index 2001h stores two 8-bit values:** Index 2001h, Subindex 00h = 2 Index 2001h, Subindex 01h = first 8-bit value Index 2001h, Subindex 02h = second 8-bit value

The following table represents the organization and structure of the Object Dictionary:

Index Range	Description
0000h	Reserved
0001h-0FFFh	Data Types
1000h-1FFFh	Communication Entries
2000h-5FFFh	Manufacturer Specific
6000h-9FFFh	Device Profile Parameters
A000h-FFFFh	Reserved

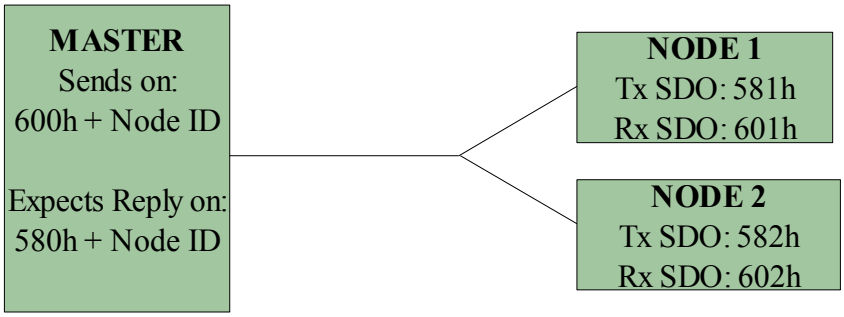
2.3 Accessing the Object Dictionary

To access the object dictionary, the CANopen message identifier “Service Data Objects (SDO)” is used.

To send an SDO message, you must send a CAN message. There are two reserved message IDs that are used: one for sending data to a specific node and one for responses that were sent by that node.

Default Message IDs for SDO Communication

Messages are sent using the message ID: 600h + Node ID
The expected reply uses the message ID: 580h + Node ID



An SDO message contains eight bytes of data, of which, the first byte is used as a specification byte. In the specification byte, the bits are primarily used to specify whether the message contains a read, write, or abort (error) message. The additional bits in the specification byte are used to indicate if the message is an “expedited transfer” or a “segmented transfer.”

An “expedited transfer” is a message where all the data that is exchanged is a part of one message. A “segmented transfer” is a data transfer where all the data can not fit into one message and must be sent in multiple messages.

Bytes two through four contain the “multiplexer” which is a combination of a 16-bit index and an 8-bit subindex that identify the Object Dictionary entry that will be accessed. The byte order of the “multiplexer” is: the low byte of the 16-bit index, the high byte of the 16-bit index, and finally the 8-bit subindex.

Bytes five through eight are used to transmit data. This data is transmitted in either an “expedited” or “segmented” fashion, as described above.

SDO Message

Byte 1	**Bytes 2 - 4**			**Bytes 5 - 8**			
Specification Byte	Multiplexer			Data			
Byte Containing the Read, Write, or Abort Message, and if the Transfer Type is Expedited or Segmented	Low Byte of the 16-Bit Index	High Byte of the 16-Bit Index	8-Bit Sub-Index	Bit - 5 DATA	Bit - 6 DATA	Bit - 7 DATA	Bit - 8 DATA

2.4 Handling Process Data

In a CAN system, it must be possible for each node to transmit data at any time and to process multiple variables into a single message. To accomplish this in CANopen, Process Data Objects (PDO) are used.

There are two separate types of PDOs: Transmit Process Data Objects (TPDO) and Receive Process Data Objects (RPDO). The slave Node sends TPDOs and receives the RPDOs. The Master will receive the TPDOs

and sends the RPDOs.

2.4.1 Transmit Process Data Objects (TPDO)

CANopen supports multiple communication methods that allow nodes to transmit data individually (event or time based), through individual or group polling, or a combination of methods. The four major methods of transmit triggers are:

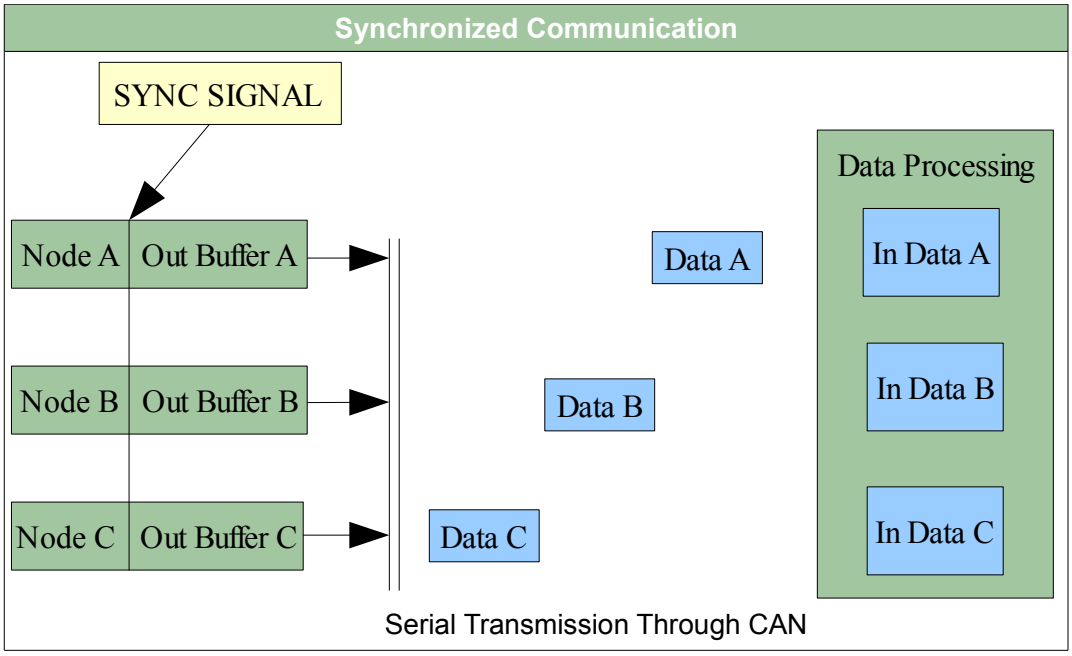
- Event Driven
- Time Driven
- Individual Polling
- Synchronized, or Group Polling

The **Event Driven** transmission method transmits a TPDO message every time the process data changes. An event can be anytime that the data changes, or only a certain change in the data. The Device Profile specifies what exactly the event is. Keep in mind that if there are inputs, the data will not be sent until there is a change in the data (i.e. if the node is started and an input is "on," the message will not be sent and will appear to be "off" until the input is turned off and back on again. There must be a change in the data before any information is sent).

The **Time Driven** transmission method transmits a TPDO message on a fixed time basis. The fixed time is set, in milliseconds, through the "Event Timer." If the Event timer is set to 50 milliseconds, then a TPDO will be sent every 50 milliseconds. If there are multiple nodes using an "Event Timer" set at the same time, 20 milliseconds, there is no synchronization between the nodes, resulting in the possibility of the TPDOs being sent within the same millisecond or at different times within the 20 millisecond time window.

The **Individual Polling** transmission method uses a message to trigger a node, telling it to send a message.

The **Synchronized, or Group Polling** transmission method is used to transmit multiple TPDOs to multiple nodes, at the same time. A "SYNC" signal is sent to all the nodes at once. The SYNC signal is a specific message with no data, and is used only for synchronization purposes. The signal is sent based on a timer, which represents a global timer for all nodes, as opposed to the local "event timers" that the Time Driven method uses.



2.4.2 Receive Process Data Objects (RPDO)

In a node's Object Dictionary, the Index area from 1400h to 15FFh is reserved for the RPDO communication parameters. Once these parameters have been set, the RPDOs can be received.

The parameters for each RPDO are accessible through the Subindex.

Name	Subindex	Data Type
Number of Entries	0	Unsigned 8-bit
COB ID	1	Unsigned 32-bit
Transmission Type	2	Unsigned 8-bit
Inhibit Time	3	Unsigned 16-bit
Reserved	4	Unsigned 8-bit
Event Timer	5	Unsigned 16-bit

The **Number of Entries** for a RPDO is set to 5 if an “Event Timer” is used. It is most common to not use the “Event Timer,” and in which case, the “Number of Entires” should be set to 2.

The **COB ID** is the CAN message identifier that the RPDO uses. This determines which CAN message should be received and interpreted.

The **Transmission Type** determines if the RPDO is processed immediately, or if the node needs to wait for a time or SYNC signal.

The **Inhibit Time** is not used for RPDOs. If it is implemented, it should be set to 0.

The **Reserved** parameter should not be implemented.

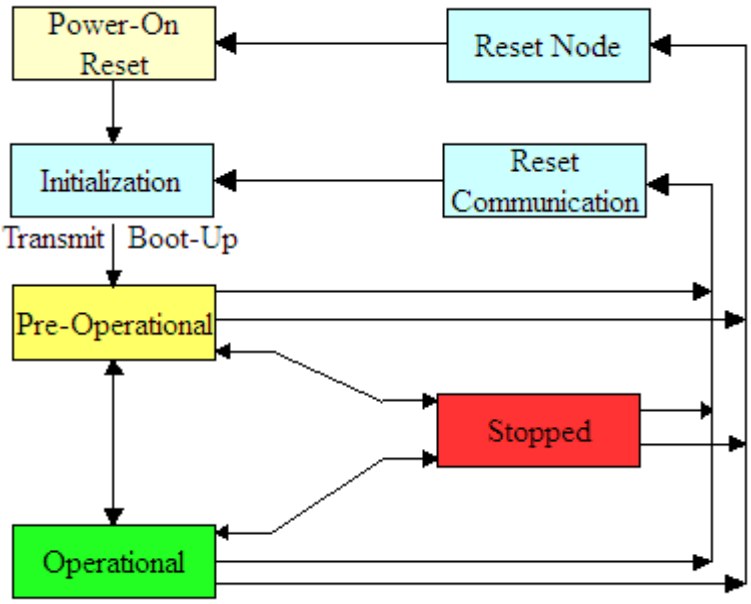
The **Event Timer** may be used to flag an emergency response if the RPDO has not been received by the time the timer has run out. The timer resets every time the RPDO is received.

2.5 Network Management (NMT)

2.5.1 Network Management States

In CANopen, every slave node must implement an Network Management (NMT) state machine. This allows the slave nodes to operate in different states. The table below demonstrates the states that the slave node can be in. Note that some of the states changes can occur automatically, but others require the node to receive an NMT Master message. The Master message can be directed at all nodes or to an individual node, and contains the new state that the node should switch to.

Network Management of Slave Node States



When a CANopen slave node is powered-up, the node comes out of the “Power-On Reset” state and goes into the “Initialization” state. In this state, the slave node initializes the application, the CAN/CANopen interfaces, and communications. The node will then attempt to transmit its boot-up message. When the boot-up message is transmitted successfully, the slave node will then enter the “Pre-Operational” state.

When the slave node has reached the “Pre-Operational” state, the NMT Master message can be used to switch the slave node between the main three states: “Pre-Operational,” “Operational,” and “Stopped.”

The NMT Master message can also trigger two reset actions: “Reset Communication” and “Reset Node.” When the “Reset Communication” message has been sent, the slave node will then reset the CAN/CANopen communication interfaces, and upon receiving the “Reset Node” command, the slave node will completely reset, including all peripherals and software. Both of the reset commands will result in a new boot-up message being transmitted by the slave node, and return it back to the “Pre-Operational” state.

2.5.2 CANopen Messages Produced and Consumed

The main difference in the NMT states is that each state allows certain CANopen communications. The following table shows which communications may be performed in each of the main states.

NMT State Dependant Communication				
Communication Type	**Initializing**	**Pre-Operational**	**Operational**	**Stopped**
Boot-Up	✔			
SDO		✔	✔	
PDO			✔	
SYNC/TIME		✔	✔	
Emergency		✔	✔	
Heartbeat/ Node Guard		✔	✔	✔

2.5.3 Heartbeat and Node Guarding

2.5.3.1 Node Guarding

All slave nodes should implement the “Heartbeat” or “Node Guard” services. In “Node Guarding” the NMT Master polls all the slave nodes for current NMT state information. If the node does not respond within a certain time window, the Master assumes that the node has been lost and can take appropriate action (i.e. shut down or restart the system).

Using this method creates a 'safer' system in that the system becomes dependent on the Master and not the individual nodes. If the Master fails, then the system fails.

2.5.3.1 Heartbeat

The heartbeat method allows for more flexible monitoring options. Each node transmits a 'heartbeat,' which is a 1-byte CAN message that contains the NMT state of the node. A benefit of using the heartbeat method is that there is no polling required. This cuts the bandwidth that is used for monitoring in half. Another benefit of the heartbeat method is that no node is dependent of another node. In the node guarding method, all nodes are dependent of the master node, so if the master fails, then all the nodes are effected. With the heartbeat method, the failure of a single node doesn't result in the failure of all the nodes.

2.5.4 Emergencies (EMCY)

Each CANopen slave node has an emergency message. The identifier for the CAN message is 80h plus the Node ID. Each emergency message contains eight data bytes.

The first two data bytes contain the CANopen error code. The third data byte contains a copy of the error register. The last five data bytes contain specific error codes.

The emergency message is only sent once, and the message remains there until another message is sent to reset that emergency.

EMCY Message		
Bytes 1- 2	**Byte 3**	**Bytes 4 - 8**
CANopen Error Code	Copy of the Error Register	Specific Error Codes

3 Implementing CANopen

This section goes through examples on how to implement in CANopen

3.1 Getting Started

First, open CoDeSys and start a new project from template. Choose the template that matches the controller that is being used.

3.1.1 Library Management

Go to the “Resources” tab, and then click on the “Library Manager.” In the “Library Manager,” click “Additional Library,” or press the “Insert” button on your keyboard. When the dialog box opens, select “col2.lib” and click open.

3.1.2 PLC Configuration and Communications

Now, go to “PLC Configuration.” Click on the name of your controller and then click the “Module Parameters” tabs. For “LoginType”, make sure the value is set to CANopen. If you are using a controller that has multiple CAN buses (i.e. the ESX CS), the make sure that you have “CANopen via CAN1” selected.

Now the CAN communications must be configured. Expand the PLC configuration first for CAN0. Under CAN0, click on “CANMsgObj13,” then on the “Channel Parameters” tab, and set it to RXRTR. This sets up a CAN Object that will be used for Node Guarding. Now click on “CANMsgObj14,” then on the “Channel Parameters” tab, and set it to TX. This sets up a CAN Object that will be used for transmitting. Now click on “CANMsgObj15,” then on the “Channel Parameters” tab, and set it to RX. This sets up a CAN Object that will be used for Receiving. If you wish to change the message object that CANopen receive, transmit, or node guard is done on, there are global variables located in the COL2 library that can be changed.

Click on the “Online” tab on the tool bar at the top of the CoDeSys Environment. Under the “Online” tab, select “Communication Parameters.” Here be sure that you set up communication for CANopen.

3.1.3 Program Set-up

Now, go back to the “POU” tab. Right click in the object window, and add a new object. Create a new “Function Block” with a unique name (i.e. FB_CONTROL_SLAVE_1). Declare the Function Block in your PLC_PRG “VAR” section with a unique name that corresponds to the Function Block name. Then make a call to that Function Block in your program. See the example code below.

The above declaration and Function Block call are marked by red arrows.

EXAMPLE CODE

```
PROGRAM PLC_PRG
VAR
        firstcycle      :   BOOL := TRUE;
        set_relay       :   FB_SET_RELAIS;
        fbTriggerWD     :   FB_WD_TRIGGERN;
        fbControlSlave1 :   FB_CONTROL_SLAVE_1;
END_VAR
---------------------------------------------
IF firstcycle = TRUE THEN
        firstcycle := FALSE;
        (* --- turn staying alive on --- *)
        StayingAlive := TRUE;
        (* --- turn relay on --- *)
        Relay := TRUE;
END_IF;
(* --- call fbTriggerWD() cyclical to trigger the watchdog --- *)
fbTriggerWD();
(* --- call FB that starts and controls slave node 1 --- *)
fbControlSlave1();
```

Open the newly created Function Block from the “POU” tab. First, the following declarations must be made.

NOTE: These declarations allow for starting the node, setting up node guarding, and setting up receiving:

EXAMPLE CODE

```
FUNCTION_BLOCK  FB_CONTROL_SLAVE_1
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
        (* --- STATUS --- *)
        iStartStatus            : INT;
        iSendStatus             : INT;
        iConfigStatus           : INT;
        iGuardStatus            : INT;
        iGuardConfStat          : INT;
        iRXstatus               : INT;
        Start                   : BOOL := FALSE;
        (* --- MESSAGES --- *)
        dwID_Node1              : DWORD;
        bDlc_Node1              : BYTE := 8;
        abData_Node1            : ARRAY [0..7] OF BYTE;
        bHandle_Node1           : BYTE;
        bMessages               : BYTE;
```

EXAMPLE CODE

```
        (* --- NODE GUARDING --- *)
        bNodeID             : BYTE := 1;
        wGuardTime          : WORD := 20; (* in ms *)
        bLifeTimeFactor     : BYTE := 3;
        (* --- FUNCTIONS ---*)
        ConfigRX            : FB_COL2_CONFIG_RX_PDO;
        RX                  : FB_COL2_HANDLE_RX;
        HandleGuarding      : FB_COL2_HANDLE_GUARDING;
END_VAR
```

Next, in the code block, the following must be executed:

- Starting the Node
- Configure Receive Messages
- Configure Node Guarding
- Execute Receive Function
- Execute Node Guarding

Starting the Node: To start the node, a call must be made to F_COL2_NMT(bNMTCom,bNodeId). To start the node, the "bNMTCom" should be "NODE_START." The "bNodeId" parameter should be filled out with the node ID of the node that should be started. If '0' is passed through the "node" parameter, then all nodes will be started. If the node was started correctly, the function will return '0.'

Configure Receive Messages: To configure for receiving messages, a call must be made to FB_COL2_CONFIG_RX_PDO. The parameter for this function block, "dwID," is the CAN ID that the message should be received on. It will return the handle for the node, "bHandle," and a status "status."

Configure Node Guarding: To configure node guarding, a call must be made to F_COL2_CONFIG_GUARDING(bNodeId, wGuardTime, bLifeTimeFactor). The parameter "bNodeId" is the Id of the node that will be guarded. The "wGuardTime" parameter sets the distance of guarding messages, in milliseconds. The "bLifeTimeFactor" parameter sets the number of missed guardings that will cause detection of a "dead node" The function returns a status.

Execute Receive Function: To receive messages from the node, the function block FB_COL2_HANDLE_RX must be called. When you call this function block, all new data received in the messages will be stored in "atCOL2RXPDOs."

Execute Node Guarding: To start node guarding, a call must be made to the function block FB_COL2_HANDLE_GUARDING. The function block automatically handles the guarding and will return two values; "status" and "qNewProblems." The status of the node guard is stored in aeCol2GuardingStatus.

Example Code: The example code below implements all of the above function calls. On the first execution, when boolean "Start" is false, the node is started, the necessary configurations for node guarding and receiving messages is made, and the boolean "Start" is set to true, so that the program will not execute this initialization phase again. After the initialization phase is complete, the program will begin receiving messages, and node guarding automatically.

The receive messages function will begin to receive all new messages . Keep in mind that nothing will happen until there has been a change in data, or the methods are changed. See the SDO and PDO sections above.

The node guarding function, runs with the parameters that were configured. In this case the guard time is set to 20 ms, and the life time factor is set to 2. So the guard checks every 20 ms and if there are 3 or more errors. If there is a problem, the status of the node is checked. If the node is "dead" then the node is immediately restarted. See the Node Guarding section above for more information on node guarding.

EXAMPLE CODE

```
(* --- On First Execution: Turn NODE1 'ON,' Configure RX, Configure
NODE GUARDING --- *)
IF Start = FALSE THEN
   iStartStatus := F_COL2_NMT(NODE_START,1);
   ConfigRX(dwID := 16#181, bHandle => bHandle_Node1, status =>
   iConfigStatus);
   iGuardConfStat := F_COL2_CONFIG_GUARDING(bNodeID,
   wGuardTime, bLifeTimeFactor);
   Start := TRUE;
ELSE
   (* --- RECEIVE MESSAGE --- *)
   RX(Status => iRXstatus, bNewMessages => bMessages);
   (* --- GUARDING --- *)
   HandleGuarding();
END_IF;


(* --- If There is a New Problem --- *)
IF HandleGuarding.qNewProblem THEN
  (* --- If the Status of the Node is Dead --- *)
   IF aeCOL2GuardingStatus[1] = COL2GuardingDead THEN
      (* --- RESTART the Node --- *)
      F_COL2_NMT(NODE_START, 1);
   END_IF;
END_IF;
```

3.2 Reading an Input from a Node

After the set-up code has been written, it is very simple to read an input from the node. When a message is received from the node, it is stored in and can be read from "atCOL2RXPDOs[n]." The following code sets DigitalOut1 of the Master Controller with the information read from the input of the slave node.

EXAMPLE CODE

```
(* --- Set DigitalOut1 of the Controller from the Input1 of the Node
--- *)
IF atCOL2RXPDOs[1].abData[0].0 = 0 THEN
        DigitalOut1 := 0;
ELSE
        DigitalOut1 := 1;
END_IF;
```

4 Troubleshooting

This section is for basic troubleshooting.

4.1 Check the CAN Bus

Be sure that your CAN Bus is connected correctly and that it is terminated correctly. You can reference the CAN Bus application note (AN 2001: Troubleshooting CAN Bus) for explicit information regarding troubleshooting the CAN Bus.

4.2 PLC Configuration

Often times, a problem occurs because something in the PLC Configuration was incorrectly set or not set at all. Check section *3.1.2 PLC Configuration* for specific instructions on PLC Configuration Set-up.

4.3 Node Error

When you initialize a node, a status is sent. Check the CAN messages, if you are not receiving a response, or you are receiving an error message, then your system will not work correctly.

If you coded status checks into your program, you can easily check the status code for errors.

Check the emergencies (EMCY). There may be something causing an emergency that is affecting your system.

5 Summary

CANopen uses CAN communications to transmit information. It uses an Object Dictionary to store all of the information that can be used. You can access the Object Dictionary using Service Data Objects (SDO) and process data can be sent through Process Data Objects (PDO). Node Guarding can be used as safety precaution and/or a way to ensure that the system is working correctly.

6 Resources

Title	Author(s)	Published
Embedded Networking with CAN and CANopen	Olaf Pfeiffer, Andrew Ayre, and Christian Keydel	2003, RTC Books
COL2 Help File	STW	Delivered with CoDeSys v2.3 Install
CoDeSys Users Guide	STW	Delivered with CoDeSys v2.3 Install

7 History

Revision	Date	Author	Comments
1	07/15/2008	J. Yore	Created the Application Note
1.1	07/17/2008	J. Yore	Added “Heartbeat” Section and Edited Content
1.2	08/21/2008	A. Jansen	Internal Changes

STW Technic, LP
3000 Northwoods Pkwy.
Suite 260
Norcross, GA 30071

Phone: (770) 242-1002
Fax: (770) 242-1006

About Us

STW Technic is the premier manufacturer of mobile electronics for on- and off-highway vehicles. A wholly owned subsidiary of STW GmbH, Germany, STW Technic is located in Atlanta, GA.

STW was founded in 1985, and has since provided electronic controls for world wide market leaders of agriculture, construction, municipal and military vehicles as well as many other kinds of mobile equipment. In 2007, STW will sell about 60,000 freely programmable controllers (more than any other manufacturer) in more than 200 different variations into these markets.

Due to highly demanding safety requirements for many applications in mobile equipment (i.e. cranes, fire equipment, etc.), many of STW's controllers are certified based on IEC 61508 (SIL2) and EN 954-1 (Cat. 3) standards. STW is also ISO 9001 certified, and further certification includes ISO/TS 16949:2002, the quality standard of the automotive industry.

In addition to the controller product range, STW offers displays, joysticks, sensors, and other electronic components to provide a complete electronic system for vehicles.

STW is also a supplier of robust pressure and force measurement sensors with thin-film, ceramic or silicon technology. STW specializes in applications in extreme conditions, which includes pressures up to 3,000 bar (44,000 psi) and media temperatures up to 300 °C (540 °F).

STW is a reliable partner, who not only supplies controllers, but can also train you to develop your own applications, write the applications for you, maintain inventory for you, and do everything a control engineering department would do.