



KoliadaES

**EtherDATA – A Distributed Data System
for Small Embedded Systems**

An Overview

Revision History

12/26/2013 – First release

4/25/2014 – Updates for CC8051 release

8/12/2014 – Additional changes for CC8051

Table of Contents

<i>KoliadaES</i>	1
EtherDATA – A Distributed Data System for Small Embedded Systems	1
An Overview	1
Introduction	4
Purpose	4
Scope	5
Acronyms	5
References	5
Distributed Data Objects	5
Description	5
Distributed data objects	5
Types	6
Data Access	7
Data Replication	7
Fragmentation	8
Allocation	8
Encoding	8
Security	8
Examples	8

Introduction

Purpose

A distributed data system is any system of communicating nodes that each maintains data objects that need to be accessed as a single, coherent, data system.

KoliadaES Distributed Data System (EtherDATA) is one such implementation designed for the transparent access and management of distributed and replicated data objects on small embedded devices.

EtherDATA is designed to abstract data references for small, heterogeneous systems. In an 8051, EtherDATA can be deployed in as few as 5k bytes ROM and negligible RAM footprint (< 128 bytes).

EtherDATA addresses the following architectural precepts;

Logical Data Independence

EtherDATA data objects are described and accessed via an external schema.

Physical Data Independence

EtherDATA data objects are stored and referenced by a logical to physical mapping defined by the system and hidden from the application.

Network Transparency

EtherDATA can be implemented over a variety of networking technologies including, but not limited to TCP/IP, Ethernet and Wireless using a variety of node architectures. The details of communication and node architecture are hidden from the application.

Replication Transparency

Data replication improves the locality of reference and improves the redundancy of access in the event of failure. EtherDATA data objects are transparently replicated across all the nodes, or groups of nodes, in network.

Fragmentation Transparency

Small-embedded systems do not have the resources to store all the data accessible by the system and any database must be fragmented across the network nodes. EtherDATA hides this fragmentation from the application.

EtherDATA is a data *definition, storage and access* system that addresses the additional needs of replication across the data system and a simple data definition paradigm.

EtherDATA does not provide relational database management facilities.

Scope

This document is designed to guide a developer in implementing an application using EtherDATA as might it be used with data communication architectures over Ethernet or wireless (RF).

Acronyms

References

Distributed Data Objects

Description

EtherDATA objects are described by the KoliadaES Data Description Language (kDDL).

kDDL is a general-purpose data description language that can be used to encode data objects for both ‘managed’ and ‘unmanaged’ purposes. Managed objects are defined to be a coherent part of a distributed data system. Unmanaged objects are objects that are formally described and may be easily shared between systems but are not part of a distributed data system.

For the purposes of this document we will focus on managed data objects only.

Distributed data objects

All data objects must be described within a data object description file. Each data object is fully described in a manner similar to a MIB by a set of object attributes using the syntax detailed in Appendix 1 and broadly;

modifier type *name* { attribute *value*; ... }

Where;

modifier	- is a type modifier
type	- is a fundamental or derived object type
name	- is the defined name of the object
type	- is an optional field type identifier
attribute	- is an attribute identifier
value	- is the attribute value appropriate to the attribute identifier. A value has either type ‘ <i>type</i> ’ or ‘ <i>string</i> ’.

An object name must be unique within the group the object is defined in. Attributes may be defined in any order and are simply key/value pairs.

EtherDATA currently defines the following *required* attribute fields;

access – is the type of access to be allowed for this object

EtherDATA currently defines the following *optional* attribute fields;

group – defines any replication groups object is part of

status – defines the status of this object

These attributes are described more fully in following sections. Other attribute fields are optional and application specific.

Objects may be defined as either static or dynamic object arrays thus;

```
type name[] { field value; ... }    // dynamic object array of type
type name[n] { field value; ... }   // static object array of type
```

Where n is a positive integer. The support of dynamic object arrays is implementation specific.

Data Objects attributes are encoded for the system such that each object is unique across the system

Types

For example, a data object may be described as one of a set of ‘fundamental’ types;

char - a character (system dependent size)
int - signed integer (system dependent size)
word - unsigned int (system dependent size)
byte - an unsigned 8 bit char
wyde - an unsigned 2 byte word
teta- an unsigned 4 byte word
real - a floating point number (system dependent size)
string - a zero terminated set of asciiz chars
bool - logical true/false (system dependent size)

Int8 - signed 8 bit word
Int16 - signed 16 bit word
Int32 - signed 32 bit word
Int64 - signed 64 bit word
UInt8 - signed 8 bit word
UInt16 - signed 16 bit word
UInt32 - signed 32 bit word
UInt64 - signed 64 bit word
Real - double precision floating point value
String - a zero terminated set of asciiz chars
Bool - signed 8 bit word

For types with system defined sizes, sizes must be set by either by DDL or compiler build options and must be consistent across the system.

Data types may be modified using ‘global’, ‘local’, or ‘remote’. Global data objects are replicated globally (across an EtherMESH), local data objects are only defined on the ‘owner’ system but may be accessed by remote systems. Remote data objects are declared locally but defined remotely. This allows a node to access a remote object without the need to carry the data definition. In this case, the definition is retrieved from a remote node and cached locally. Remote definitions allow a small node to access a much larger data set than might fit on a small system. Remote capabilities are implementation dependant.

Whereas constructed types (class types) are in prototype, these are not currently available in release.

Data Access

EtherDATA objects are accessed using a set of data access primitives, `dbGet()`, `dbSet()` and `dbSubscribe()`. Each of `dbSet` and `dbGet` have array index variants `dbGetArray()` & `dbSetArray()`. `dbSubscribe()` allows a node to ‘subscribe’ to an object in such a way as to receive a callback for any change to the object value. Subscription to an indexed object will receive one callback for each change to any object index. The call back will refer only to the object index that changed.

EtherDATA uses a naming convention to disambiguate data objects based on the group within which the data object is found. Thus a data object may be referenced as “house.kitchenTemp” to refer to a group called ‘house’ that replicates a data object value called ‘kitchenTemp’. Group names are unique and currently support only one level of re-direction. Object names are unique across a group. Where the object is not defined as part of a group the device name may be substituted to allow reference to the object on a specific device. In this way, an object may be defined on multiple devices but with a value that is specific to that particular device.

Device and group name constructions are implementation specific and typically map to the underlying communications protocols. For example, EtherDATA over EtherMESH maps the device and group names directly to EtherMESH device and group names.

Data Replication

EtherDATA supports transparent data replication across a distributed network. Replication is by group id and thus object names must be unique within that group. A data object may be referenced by access to any device currently supporting that group. Typically, groups are used to transparently replicate data from one set of nodes to another such that any given access to the device will be local rather than remote.

For example, a gateway device might make certain data available locally such that external queries hit the gateway only and are not propagated into the network. Another example, a gateway or specific logging device may be used to track data over a period of time. This device then makes its aggregated data available to the network as an indexed array or other derived data object.

In another example, a group may be used to provide access redundancy. A group transparently replicates data items and thus only needs a minimum of one device in a group to be ‘live’ to service a dbGet request.

Fragmentation

Currently, EtherDATA requires that any object fit in the maximum payload size offered by the underlying transport.

Allocation

EtherDATA can be configured to use either ram, flash or file storage. Storage is selected at compile time and cannot be changed at runtime. Different devices may use different storage paradigms, but they cannot be mixed *within* the device.

Encoding

All data objects are encoded using ASN.1 (BER) for both storage and communication. This significantly reduces the space required to save and send objects.

Security

EtherDATA is ‘security blind’ and depends on the underlying security of the system on which it is deployed. EtherDATA does not manage authentication or access control (other than as may be defined by the ‘access’ object field – read, write, none).

Examples

There are a number of examples available in the SDK that show the execution of most EtherDATA features.

Appendix 1 – Summary EtherDATA data description

kDDL uses the following keywords;

<i>char</i>	- a character (system dependent size)
<i>int</i>	- signed integer (system dependent size)
<i>word</i>	- unsigned int (system dependent size)
<i>byte</i>	- an unsigned 8 bit char
<i>wyde</i>	- an unsigned 2 byte word
<i>teta</i>	- an unsigned 4 byte word
<i>real</i>	- a floating point number (system dependent size)
<i>string</i>	- a zero terminated set of asciiz chars
<i>bool</i>	- logical true/false (system dependent size)

<i>Int8</i>	- signed 8 bit word
<i>Int16</i>	- signed 16 bit word
<i>Int32</i>	- signed 32 bit word
<i>Int64</i>	- signed 64 bit word
<i>UInt8</i>	- signed 8 bit word
<i>UInt16</i>	- signed 16 bit word
<i>UInt32</i>	- signed 32 bit word
<i>UInt64</i>	- signed 64 bit word
<i>Real</i>	- double precision floating point value
<i>String</i>	- a zero terminated set of asciiz chars
<i>Bool</i>	- signed 8 bit word

access
readonly
writeonly
readwrite
group

<i>global</i>	- type modifier indicating ‘global’ scope
<i>remote</i>	- type modifier indicating remotely defined context
<i>local</i>	- type modifier indicating locally defined context

Comments follow the style of C/C++ comments using // delineating comments to the end of the line and /* .. */ delineating block comments which may also be nested thus; /* ... /* ... */ */

A string literal is any sequence of chars enclosed in double quotes thus; “This is a string”

An identifier is any valid c/c++ identifier.

The following is an example of valid kDDL content;

```
// comment
/* comment /* nested */ */

// values may be declared local, remote or global
// unmarked values are local by default

// all local and global values must have a definition section that
// describes value attributes
// remote values may have a definition section but do not require it as
// long as dbGetDefinition() is available

remote kitchenTemp      // a value maintained on a remote node
{
    // (no local definition)
    // declare attributes we use so attribute enumeration can work
    // we do not need to declare all attributes, just the ones we use
    description,
    units
}

global int bodyTemp      // a value maintained globally
{
    description "Grandma's current body temp";
    access readonly;

    units "°C";
    max 39;
    min 33;

    // expressions may be used to map to named function handlers
    (value > max) tempAlert;
    (value < min) tempAlert;
}

bool led[3] // a value maintained locally
{
    // local values are not globally unique
    description "Device LEDs 0 - red, 1 - blue, 2 - green";
    access readwrite;

    (value == true) ledAlert;
    (value == false) ledAlert;
}
```

Note that any attribute that is not a formal attribute (access, group) is a simple key/value pair such that the key can be any identifier and the value any value of the same type as the data object or a string.