

ACCESS.bus Primer

By Randy Goldberg

This application note is intended to be a simple introduction to the ACCESS.bus (ACCESS.bus is a block similar to the Phillips 8584 I²C controller). It will explain the ACCESS.bus and provide examples of how code should be structured to correctly transfer data to an ACCESS.bus peripheral such as a ACCESS.bus EEPROM. This application note will focus on the FDC37C95xFR chip, but the information herein is easily applicable to our other chips with embedded ACCESS.bus controllers (FDC37C93xFR, and FDC37C93xAPM). Our ACCESS.bus block is based on the Phillips 8584 I²C controller.

Please be advised that a separate application note (AN 6.17) explains the differences between SMB (system management bus), ACCESS.bus and I²C.

ACCESS.bus Basics

ACCESS.bus is a low speed (Max bit rate 100k), byte oriented two wire serial bus which can have multiple devices attached to it. It is intended primarily for applications in which low cost and low speed requirements are dominant. An example of a typical application would be a laptop with an FDC37C95xFR which has a serial EEPROM, smart battery, temperature sensor and A/D converter all connected to the ACCESS.bus. The FDC37C95xFR's 8051 could have connectivity to many devices all with only two pins being utilized.

In a ACCESS.bus system, all ACCESS.bus devices must have a unique address. A specific ACCESS.bus device must act as a Master/Slave device (i.e. an FDC37C95xFR) or a slave only device (i.e. a serial EEPROM). Multiple masters can co-exist in a system, but only one master is in control of the bus at any one time. A Master/Slave device will switch to slave mode if another master in the system addresses it.

A simplistic overview of an ACCESS.bus transaction is a master device places a start character on the Access bus with the address of the slave and the direction of the desired data transfer. If a slave recognizes its address and is ready to transfer data, it will acknowledge. After the valid start, the host reads or writes to the master's data register which will cause data to be transferred across the ACCESS.bus. The Master will then place a stop condition on the bus. Only the master can place a stop condition on the bus, if a slave wishes to terminate the transfer it must stop acknowledging individual transfers so that the master will generate a stop. The two main types of transfers include master TX to slave RX and master RX to slave TX. Note that a master device can become a slave device only if addressed by another master in the system.

The ACCESS.bus interface is only two wires (1 clock and 1 data line, both open drain). Each byte transferred has 9 clock slots. In the first 8 clocks slots the transmitting device places data on the data lines, in the 9th slot, the receiving device puts an acknowledge on the data line (only if it has properly received the data). The data line should not transition during the clock high time when the byte or acknowledge is being transferred. Data line transitions when the clock is high represents start and stop conditions on the ACCESS.bus. A high to low transition of the data line when the clock is high is the 'start' condition. A low to high transition of the data line when the clock is high is the 'stop' condition. The master device always generates the clocks and the start and stop conditions, regardless of the direction of data transfer. The slave device can slow the transfer by holding the clock line low. This allows devices with slower maximum speed to communicate with faster devices (although at a slower data rate).

Software Examples

Consider the case of an FDC37C95xFR in a system with an ACCESS.bus 24c02 EEPROM (our FDC37C95xFR demonstration card has an ACCESS.bus serial EEPROM on it). Code examples shall be given that show how to execute single byte read and write transfers as well as multiple byte transfers. Examples will also be given that show how to search for EEPROMS in the system as well as how to do 'ack polling' of the EEPROM after a write to determine when it is has finished the write.

Before getting into the code, a quick discussion of how the ACCESS.bus is embedded in the FDC37C95xFR. The FDC37C95xFR has an ACCESS.bus controller built into it that is accessed through 8051 memory mapped registers 7f31-7f34. 8051 code can be generated to control the ACCESS.bus block. But since C code running on a PC host is considered easier by the author to write than 8051code, a lab workaround was used to exercise the ACCESS.bus block from the host side. It is called the Mailman utility, and it is available to our customers and provides an easy way to get access to the logical blocks that are connected to the 8051 through the host interface. The 8051 communicates with the host through a number of mailbox registers that are accessible by the host and the 8051. Generic multipurpose 8051 code was written that acts as a 'mailman', transferring data from the mailbox registers to the desired 8051 register. Once 'mailman' code is running, host software can easily control 8051 devices. This mailman code is available to our customers and these code examples will show how to use it from the host point of view (all the FDC37C95xFR demonstration cards are shipped with mailman code programmed in the flash ROM). Of course the code examples can be used as a template to generate 8051 code.

Overview of Code Example

The code example that is provided with this application note is intended to run on SMSC's demonstration card that has the 8051 running out of a flash ROM that has mailman code running. Our demonstration card also contains a 24c02 ACCESS.bus EEPROM that will act as the slave device. Although this code was written for the demonstration card, it should work on any platform that has the FDC37C95xFR's 8051 running mailman code, and a ACCESS.bus EEPROM. The code itself reads and writes the EEPROM through the FDC37C95xFR's ACCESS.bus block utilizing byte and packet type transfers. The code does not really do anything except provide an example of how ACCESS.bus code can be structured.

Here is a quick flow of the program:

```
Find and configure the FDC37C95xFR
Check the Mailman and make sure it is running
Initialize the ACCESS.bus
Search for EEPROMS
BYTE write some data to the EEPROM
BYTE read some data from the EEPROM
Packet write some data to the EEPROM
Packet read some data from the EEPROM
```

The items in boldface will be presented in Pseudo code so they can be thoroughly explained, as well as pointing out some subtleties. Also documented will be a brief explanation of all the procedures that the code example uses.

Register/Protocol Information

Registers

Bit by bit registers descriptions of the ACCESS.bus can be found in the FDC37C95xFR data sheet. Here is a quick summary of them:

ACCESS.bus status register (7f31) read:	Status information such as the PIN bit which is the interrupt
ACCESS.bus control register (7f31) write:	Commands such as 'start' and 'stop' are sent through this register
ACCESS.bus own address (7f32) write:	The value loaded into this register is the address of the ACCESS.bus controller when being addressed.
ACCESS.bus data (7f33) write/read:	Data is passed through this register.
ACCESS.bus clock register (7f34) write:	Sets up the ACCESS.bus speed. This register also contains the soft reset bit.

Protocol

Information about the protocol can be obtained in the ACCESS.bus 3.0 specification.

Pseudo Code

Search for EEPROMS

One thing that system software can do on power up, or as part of a diagnostic routine, is survey the system for ACCESS.bus slave devices. A slave device address is made up of the upper bits which define the type of ACCESS.bus device (EEPROM, LCD driver....) and the lower address bits which define a unique address for that particular slave device (so that multiple devices of the same type can be in a system). In my example, ACCESS.bus EEPROMS with addresses between 0-7 are searched for (the last valid EEPROM found will be the EEPROM used for read and write transfers). This search can of course be extended to search for all expected devices in any given system.

The way a particular device is searched for is do a simple start command with that device being addressed. If that device is in the system it will respond with an acknowledge. This is then reflected in the status register. Below all 8 EEPROM addresses are searched for.

For EEPROM address 0-7

Out Data register, 1010 A2 A1 A0 0	Write data register with address of EEPROM that is currently being looked for. The 1010 represents the EEPROM device code, A2, A1, and A0 represent the particular device address, and the 0 indicates that this is a write command even though no data gets passed
------------------------------------	---

Out Control register, 45h	Issues the start command.
---------------------------	---------------------------

Poll bit 7 of status register and wait for it to go low. This is the PIN bit, when it goes low it indicates that the byte has been transferred and the other bits in the status register are now valid.

Check bit 3 of status register, this is the LRB (last received bit) bit. Because the last received bit is the Acknowledge bit from the slave (acknowledges are active low), this bit will be a zero if the slave at address A2A1A0 was present.

If bit 3 of status =0	Then Print "EEPROM found at address", A2A1A0
If bit 3 of status =1	Then Print "EEPROM not found at address", A2A1A0

Out command register, 42h:	Write stop command
Out data register, 00h:	Write dummy data to issue stop

Next address

BYTE Write

One of the simpler things to do with ACCESS.bus is a single byte EEPROM write. To do this, A start command is issued with the desired EEPROM address (address of the individual EEPROM) and the direction bit set to a "0" to indicate a write. A start is issued by writing the address byte to the data register and the start command to the control register. Next the desired EEPROM location to be written is sent, followed by the desired write data. Then a stop command issued. At this point the desired address and data have been transferred to the EEPROM, but this data has not been stored in the EEPROM yet. This can take several milliseconds per write byte. A small delay can be inserted in the code, or EEPROM ack polling can be used. When the EEPROM is actually writing data to its memory, it will not respond to start commands, the host could keep issuing starts and wait until the EEPROM responds with an acknowledge indicating it is done with the EEPROM write. The EEPROM ack polling routine will be described in the summary of procedures (poll_EEPROM_ACK_for_idle).

Out Data register, A2 A1 A0 0 Write data register with address of EEPROM that is currently being 1010 used. The 1010 represents the EEPROM device code, A2 A1 A0 represent the specific address of the EEPROM that is being written, and the "0" indicates that this is a write command.

Out Control register, 4ch: Issue start command

Poll bit 7 of status register and wait for it to go low. This is the PIN bit, when it goes low it indicates that the byte has been transferred and subsequent bytes can be transferred.

Out Control register, 49h: Reset start bit. This is so the next time a byte is transferred it does not re-issue a start command.

Out Data register, address: Send the data address of desired EEPROM location to be written.

Poll bit 7 of status register and wait for it to go low.

Out Data register, write_data: Send the write data

Poll bit 7 of status register and wait for it to go low.

Out command register, 42h: Write stop command
Out data register, 00: Write dummy data to issue stop

Poll bits 7,0 of status register and wait for them to go high. These are the PIN bit and bus busy bit. When they both go high it indicates a ACCESS.bus idle. This should occur after a stop is issued.

Poll EEPROM for ack. This will be described later and is just a routine to wait for the EEPROM to finish the actual write of the data to its memory. See poll_EEPROM_ACK_for_idle in summary of procedures.

BYTE Read

Slightly more complicated than a byte write is a byte read. The reason for this is that a write must first be done to the ACCESS.bus EEPROM which tells the EEPROM what location is to be read, then a read is done to fetch the data.

To do a byte read, a start command is issued with the desired EEPROM address (address of the individual EEPROM) and the direction bit set to a "0" to indicate a write. Next the desired EEPROM location to be read is sent, followed by a stop command. At this point the desired address has been sent to the EEPROM. Next a start command is issued with the desired EEPROM address (address of the individual EEPROM) and the direction bit set to a "1" to indicate a read. A dummy read of the data register should be done. This causes the desired EEPROM data to be transferred across the ACCESS.bus. The next read of the data register will contain the desired EEPROM data. A stop can then be issued.

Out Data register: Write data register with address of EEPROM that is currently being 1010 A2 A1 A0 0 used. The 1010 represents the EEPROM device code, A2 A1 A0 represent the specific address of the EEPROM that will be read, and the 0 indicates that this is a write command.

Out Control register, 4ch: Issue start command.

Poll bit 7 of status register and wait for it to go low. This is the PIN bit, when it goes low it indicates that the byte has been transferred and subsequent bytes can be transferred.

Out Control register,49h: Reset start bit. This is so the next time a byte is transferred it does not reissue a start command.

Out Data register, address: Send the data address of desired EEPROM location to be read.

Poll bit 7 of status register and wait for it to go low.

Out command register, 42h: Write stop command. Address send is complete.
Out data register, 00h: Write dummy data to issue stop

Poll bits 7,0 of status register and wait for them to go high. These are the PIN bit and bus busy bit. When they both go high it indicates a ACCESS.bus idle. This should occur after a stop is issued.

Out Data register, Start command data, Note the least significant bit is a 1010 A2 A1 A0 1 1, indicating a read command.

Out Control register, 4dh: Issue start command

Poll bit 7 of status register and wait for it to go low.

Out Control register,40h: Reset start bit. This is so the next time a byte is transferred it does not reissue a start command. Also turn off the ack bit so that when we receive the next byte of data across the ACCESS.bus we will not issue a ACK so the EEPROM will know it is the last byte the master needs transferred.

In data register: This is a dummy read that causes the real data to be transferred across the ACCESS.bus. The host should discard this data.

Poll bit 7 of status register and wait for it to go low. Also look for ACK bit to be inactive (high) because the ack bit in the control register was previously turned off.

Read_data= In data register: This is the desired EEPROM read data. This read does not cause another ACCESS.bus transfer because the ACK bit has been turned off before the last Xfer.

Out command register, 42h: Write stop command..

Out data register, 00: Write dummy data to issue stop

Poll bits 7, 0 of the status register and wait for them to go high. These are the PIN bit and bus busy bit. When they both go high it indicates an ACCESS.bus idle. This should occur after a stop is issued.

Packet Write

Writing a packet of data to the EEPROM is only slightly more complicated than writing a single byte. The 24C02 has a front end FIFO that is capable of storing 16 bytes of data. Once the stop command is issued the 24C02 will do the actual write of the data to the EEPROM. For this example a 16 byte data transfer will be done.

First send a start command with the desired EEPROM address (address of the individual EEPROM) and the direction bit set to a 0 to indicate a write. Next the starting address of the block of 16 addresses is sent followed by the 16 bytes of data, followed by a stop command. Ack polling of the EEPROM can then be done to determine when the EEPROM is done writing the data.

For the following example, assume that the write data is in a buffer called write_buffer [16]

Out Data register: Write data register with address of EEPROM that is currently being 1010 A2 A1 A0 0 used. The 1010 represents the EEPROM device code, A2 A1 A0 represent the specific address of the EEPROM that is being written, and the "0" indicates that this is a write command.

Out Control register, 4ch: Issue start command.

Poll bit 7 of status register and wait for it to go low. This is the PIN bit, when it goes low it indicates that the byte has been transferred and subsequent bytes can be transferred.

Out Control register, 49h: Reset start bit. This is so the next time a byte is transferred it does not re-issue a start command.

Out Data register, address: Send the data address of desired EEPROM location to be written (first of the block of 16)

Poll bit 7 of status register and wait for it to go low.

```
For (j=0; j<16; j++) {
  Out Data register, write_buffer [j] send the data to be written.
```

Poll bit 7 of status register and wait for it to go low.

}

Out command register, 42h: Write stop command
Out data register, 00h: Write dummy data to issue stop

Poll bits 7,0 of status register and wait for them to go high. These are the PIN bit and bus busy bit. When they both go high it indicates a ACCESS.bus idle. This should occur after a stop is issued.

Poll EEPROM for ack. This will be described later and is just a routine to wait for the EEPROM to finish the actual write of the data to its memory. (see poll_EEPROM_ACK_for_idle in summary of procedures).

Packet Read

Slightly more complicated than a byte read is a packet read. It is basically the same as a byte read except that multiple reads of the data register are done to receive multiple bytes of EEPROM data. What makes the packet read slightly more complex, is that the ACCESS.bus master, in this case the FDC37C95xFR, must stop acknowledging the receive bytes, one byte before the last byte is read. This lets the ACCESS.bus EEPROM know it is not required to provide any more data.

There is no limit to the packet size for read as there is for write (16 bytes max due to the FIFO). To make the program symmetrical, for the following example, assume that 16 bytes of data will be read, and that the read data ends up in a buffer called read_buffer [16].

Out Data register: Write data register with address of EEPROM that is currently being 1010 A2 A1 A0 0 used. The 1010 represents the EEPROM device code, A2 A1 A0 represent the specific address of the EEPROM that will be read, and the 0 indicates that this is a write command.

Out Control register, 4ch: Issue start command

Poll bit 7 of status register and wait for it to go low. This is the PIN bit, when it goes low it indicates that the byte has been transferred and subsequent bytes can be transferred.

Out Control register,49h: Reset start bit. This is so the next time a byte is transferred it does not re-issue a start command.

Out Data register, address: Send the data address of desired EEPROM location to be read (First of the block of 16).

Out command register, 42h: Write stop command
Out data register, 00: Write dummy data to issue stop

Poll bits 7,0 of status register and wait for them to go high. These are the PIN bit and bus busy bit. When they both go high it indicates a ACCESS.bus idle. This should occur after a stop is issued.

Out Data register: Start command data, note the least significant bit is a 1, indicating 1010 A2 A1 A0 1 a read command.

Out Control register, 4dh: Issue start command

Poll bit 7 of status register and wait for it to go low.

Out Control register, 49h: Reset start bit. Also leave on the ACK, because we will want additional bytes of read data.

In data register: This is a dummy read that causes the real data to be transferred ACCESS.bus. The host should discard this data.

For (j=0;j<14;j++) {

poll bit 7 of status register and wait for it to go low.

Read_buffer[j]= Inport data register This is desired EEPROM read data.
}

Poll bit 7 of status register and wait for it to go low.

Out Control register, 40h: Turn off ack. On the next receive of data, the FDC37C95xFR will not acknowledge, so that the ACCESS.bus EEPROM can stop fetching data.

Read_buffer[14] = Inport data register This is the desired EEPROM read data.

Poll bit 7 of status register and wait for it to go low. Also look for ACK bit to be inactive (high) because the ack bit in the control register was previously turned off.

Read_buffer[15]= Inport data register This is the desired EEPROM read data.

Out command register, 42h: Write stop command
Out data register, 00h: Write dummy data to issue stop

Poll bits 7,0 of status register and wait for them to go high. These are the PIN bit and bus busy bit. When they both go high it indicates a ACCESS.bus idle. This should occur after a stop is issued.

Summary of Procedures Used in Code Example

Find_FDC37C95xFR

Searches for the configuration space of FDC37C95xFR at 3f0 and 370. Set up the variables 'cadd', 'cdata' as the configuration space address and data register pointers respectively. These are the pointers to use when in configuration mode.

Config_FDC37C95xFR

This writes configuration register 0x03 (index register) to 0x83. This enables 'open mode' which will allow access to the mailbox registers when in run mode (as opposed to configuration mode). When accessing the mailbox registers in run mode, the address and data pointers will be 0xea and 0xeb. Also write the 8051 stop clock register to 0h, this will let the 8051 start executing.

Mailman Read

This procedure writes the appropriate mailman registers to have the 8051 mailman code that is running out of flash and transfers data from an 8051 register to a mailbox register for retrieval by the host.

Mailman Write

This procedure writes the appropriate mailman registers to have the 8051 mailman code that is running out of flash and transfers data from a mailbox register to a desired 8051 register.

Check Mailman

This procedure checks that the mailman is running by reading the 8051 ID register.

ACCinit

Resets and initializes the ACCESS BUS block. Sets up the desired clock rate (24MHz input clock), writes the own address register, and issues a NOP command to get the I²C block 'going'.

Check for Slave Acknowledge

A EEPROM address is passed to this procedure. A start command followed by a stop command is issued. The acknowledge bit in the status register can be checked to see if a EEPROM with a matching address was found. If an EEPROM was found, a "1" is returned by this procedure.

Check for EEPROMS

Using the `Check_for_slave_acknowledge` procedure, all EEPROMs in the address space 0-7 are looked for. The procedure will save the value of the last found EEPROM in the variable `EEPROM_found_at_this_address`. The subsequent routines that write and read the EEPROM will use the address for the EEPROM.

Wait for ACC Int

This routine polls the status register and waits for the PIN bit to go low. This is the interrupt bit, and it will typically become active after the FDC37C95xFR has finished transferring a byte of data. This routine is used when results of the other ack status bit is unknown, such as the `Check_for_slave_acknowledge` procedure.

Wait for ACC Int and Ack

This routine polls the status register and waits for the PIN bit and the ACK bit to go low. In this case we are looking for the ACK bit to be valid after a transfer, such as during a byte write routine when data is transferred to the EEPROM and the ACK represents the EEPROM acknowledging.

Wait for ACC Int and No Ack

This routine polls the status register and waits for the PIN bit to be low and the ACK bit to be high. In this case we are looking for the ACK bit to be invalid after a transfer. This routine is used during the read procedures, when it is necessary to shut the ACK bit off in the command register before the last receive of valid data. This results in the ACK bit being high after the transfer.

Poll EEPROM ACK for Idle

This routine is used by the ACCESS.bus write routines. After an ACCESS.bus write to the EEPROM is terminated, the ACCESS.bus EEPROM takes the data from its FIFO and transfers it to the EEPROM. During this time, ACCESS.bus transfers to this EEPROM can not occur and the EEPROM will respond with a NACK if polled. This routine has the FDC37C95xFR issuing start/stop commands to the EEPROM until it responds with a valid acknowledge. The EEPROM can now be accessed again.

Byte Write ACC

This procedure does a byte write to EEPROM. The EEPROM address, and the desired location within the EEPROM to written are passed to the routine. The data to be written is placed in a variable `write_byte`.

Byte Read ACC

This procedure does a byte read from the EEPROM. The EEPROM address, and the desired location within the EEPROM to read are passed to the routine. The procedure places the EEPROM read data in a variable called `read_byte`.

Packet Write ACC

This procedure does a 16 byte write to EEPROM. The EEPROM address, and the desired starting EEPROM location is passed to the routine. The 16 bytes to be written are placed in a buffer called `write_buffer`.

Packet Read ACC

This procedure does a 16 byte read from the EEPROM. The EEPROM address, and the desired starting EEPROM location is passed to the routine. The 16 bytes that are read are placed in a buffer called `read_buffer`.

Programs Provided and How to Get Started

The following software files have been zipped into a file called "access.zip", located in the "appsoftware" directory on SMSC's FTP server. You can download this software from SMSC's Web site at <http://www.smSC.com/ftpdocs/chips.html>, or dial in to the FTP server at 516-233-4272.

- Orionen.exe** This program applies power to the new FDC37C95xFR demonstration card (FDC37C95xFR Rev. B). This newer demonstration card controls VCC to the FDC37C95xFR through writeable registers, and this program must be run before anything else if the new demonstration card is being used. Do not run this program with older Rev A demonstration cards.
- ACCSMC** This is the ACCESS.bus test program.
- ORFR57** This is an FDC37C957FR configuration program. If IICSMC, does not work, this program can be run to get some information that can be feed back to SMSC.
- ORFR58** This is an FDC37C958FR configuration program. If IICSMC, does not work, this program can be run to get some information that can be feed back to SMSC.
- INITFILE** Init file needed by configuration programs.

Both FDC37C95xFR demonstration cards come from the factory with FLASH that runs MAILMAN code. Each board also has a 24C02 on it that the FDC37C95xFR can talk to. Jumpers J50 on the early Rev A board and jumpers F29 on the more recent Rev B board control the EEPROM address. The test program will use any EEPROM address that it finds, so these jumpers do not have to be changed.

If you are using the more recent Rev B demonstration card, first run ORIONEN to enable power to the board (do not run this program with the Rev. A board). Then run ACCSMC. ACCSMC must first see the configuration space of the FDC37C95xFR, and see a working Mailman before it will do any I²C tests. If it does not see a Mailman it will take some time to timeout, but be patient and it will. If any ACCESS.bus test fails, the fail data will get written to a file called 'accbus.txt'.



80 Arkay Drive
Hauppauge, NY 11788
(631) 435-6000
FAX (631) 273-3123

Copyright © SMSC 2004. All rights reserved.

Circuit diagrams and other information relating to SMSC products are included as a means of illustrating typical applications. Consequently, complete information sufficient for construction purposes is not necessarily given. Although the information has been checked and is believed to be accurate, no responsibility is assumed for inaccuracies. SMSC reserves the right to make changes to specifications and product descriptions at any time without notice. Contact your local SMSC sales office to obtain the latest specifications before placing your product order. The provision of this information does not convey to the purchaser of the described semiconductor devices any licenses under any patent rights or other intellectual property rights of SMSC or others. All sales are expressly conditional on your agreement to the terms and conditions of the most recently dated version of SMSC's standard Terms of Sale Agreement dated before the date of your order (the "Terms of Sale Agreement"). The product may contain design defects or errors known as anomalies which may cause the product's functions to deviate from published specifications. Anomaly sheets are available upon request. SMSC products are not designed, intended, authorized or warranted for use in any life support or other application where product failure could cause or contribute to personal injury or severe property damage. Any and all such uses without prior written approval of an Officer of SMSC and further testing and/or modification will be fully at the risk of the customer. Copies of this document or other SMSC literature, as well as the Terms of Sale Agreement, may be obtained by visiting SMSC's website at <http://www.smsc.com>. SMSC is a registered trademark of Standard Microsystems Corporation ("SMSC"). Product names and company names are the trademarks of their respective holders.

SMSC DISCLAIMS AND EXCLUDES ANY AND ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY AND ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND AGAINST INFRINGEMENT AND THE LIKE, AND ANY AND ALL WARRANTIES ARISING FROM ANY COURSE OF DEALING OR USAGE OF TRADE.

IN NO EVENT SHALL SMSC BE LIABLE FOR ANY DIRECT, INCIDENTAL, INDIRECT, SPECIAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES; OR FOR LOST DATA, PROFITS, SAVINGS OR REVENUES OF ANY KIND; REGARDLESS OF THE FORM OF ACTION, WHETHER BASED ON CONTRACT; TORT; NEGLIGENCE OF SMSC OR OTHERS; STRICT LIABILITY; BREACH OF WARRANTY; OR OTHERWISE; WHETHER OR NOT ANY REMEDY OF BUYER IS HELD TO HAVE FAILED OF ITS ESSENTIAL PURPOSE, AND WHETHER OR NOT SMSC HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.